Ray Simulation of Photon in Water using Openacc

A Project Report

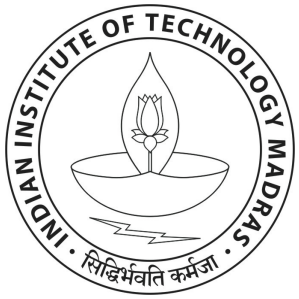
submitted by

SUBHAM AGRAWAL

EE14B059

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

MAY 2018

REPORT CERTIFICATE

This is to certify that the report titled **Ray Simulation of Photon in Water using openacc**, submitted by **Subham Agrawal**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Harishankar Ramachandran
Project Guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date: 13th June 2018

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my guide Harishankar Ramachandran for giving me an opportunity to work under him. Also I would like to thank you for constantly guiding me thoughtfully and efficiently throughout this project, giving me an opportunity to work at my own pace along my own lines, while providing me with very useful directions and insights whenever necessary.

I would also take this opportunity to thank all my friends who have been a great source of motivation and encouragement.

Finally I would also like to thank all of them who have helped me complete my project successfully.

Subham Agrawal
EE14b059
Student
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai
Date: 11th May 2018

TABLE OF CONTENTS

Contents

1 Introduction

With the growing technology, We want everything to be faster and powerful. We look for computers that has high graphics card, RAM, storage and then we look out for display and looks. Why do we want high processing graphics card? GPUs (or popularly known as Graphics Card) has a lot more cores as compared to 4 or 8 in CPUs. But as compared to processing power, CPUs are faster than GPUs. But if we look overall, the GPU wins. Now-a-days the pc games require gpu cards to do most of the processing. They can run many threads in parallel and speeds-up the process.

Talking about the disadvantage, the memory transfers from main memory to CPU is much faster than to GPU where it flows through PCI bus. Also its very difficult to program a GPU than in CPU as things get complicated when they run in parallel. Also to get speed-up, the memory transfers should be as minimal as possible.

In this project, we project 100000 (= N) rays into sea and track each ray in a time step. The main objective is to find how many rays reach the submarine (which is inside sea) in how many time steps, their standard deviation, their intensity and some other parameters. To simulate this, we are given two input files which contains the attenuation value and scattering values at different depths and angles. But we know these values at distinct angles and depths. Therefore we interpolate these curves so that the values at any angle and depths are known. This interpolation is done using UnivariateSpline function in python.

1.1 Amdahl's law

Now the question arises, how much speedup can one get. It is depicted by Amdahl's Law. Optimally, doubling the number of processing elements should halve the time taken and so on. However, very few parallel algorithms achieve optimal speedup. he potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law:

$$\text{Theoretical Speed-up} = \frac{1}{1-p+\frac{p}{s}}$$

where p is the fraction of the part that can be parallelized, s is the speed-up of the parallel execution part. Also it suggests that the maximum speedup that can be achieved is

$$\text{Maximum Speed-up} = \frac{1}{1-p}$$

2 OpenMP

OpenMp stands for Open Multi-Processing. It is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN. OpenMP is an implementation of multi-threading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. The section of the code that needs to be parallelized is marked with a compiler directive that will divide the work into different threads. Each thread has its own unique id i.e. an integer from 0. The master thread has an id of 0. We can get thread id of any thread by calling the following function: $omp\_get\_thread\_num()$. By default, each thread executes the parallelized section of code independently. We can however divide the task among different threads.

2.1 Fork-Join Model:

OpenMp uses fork-join model for parallel execution. In this model, as soon as the compiler encounters a parallel directive, it creates the specified number of threads (By default it creates the same number of threads as the number of cores in the CPU) and then joins all the threads after the work of the threads are done.

2.2 Directives and clauses of OpenMP

Now, I will mention some of the important or most used directives of openMP. The directives are placed after "$\#pragma\ omp$" and before any clause.

Directives	Use
parallel	Defines the start of the parallel region. This block will be executed by multiple threads in parallel.
for	It is a work-sharing directive which divides the work of the for loop or the iterations in the available threads.
single	This directive causes the content of the block to be run by a single thread not necessarily the master thread.
master	This directive causes the content of the block to be run only by the master thread.
critical	This directive allows only one thread to execute the content of the block at a time.
barrier	It is a forced synchronization i.e. no thread crosses this point unless all the threads have reached this point.
task	This directive divides the work into tasks and whichever thread is free picks up a task and completes it.

Clauses	Use
private	Specifies that each thread should have a own copy of that variable.
shared	Specifies that the variable is shared among all other threads
reduction	Specifies that the private variable of each thread are to be reduced as specified.
nowait	Whichever thread has completed its job can proceed further even if other threads haven't. It overrides the barrier directive.
ordered	The block executes in the same order as in the case of sequential.
schedule	This clause is only for for directive. It allows to divide the work in a static or dynamic way.

2.3 Example of Matrix multiplication

Listing 1: Matrix Multiplication

```
1 #include<omp.h>
2 #include<time.h>
3 #include<stdio.h>
```

```
4  #include<stdlib.h>
5
6  int n = 2000;
7  int arr1[2000][2000], arr2[2000][2000], ans[2000][2000],trp↩
       [2000][2000];
8  int main(int argc, char const *argv[])
9  {
10     /* code */
11
12     clock_t start, end;
13     double cpu_time_used;
14     int sum = 0;
15     double diff,t;
16     int i,j,k;
17
18     for(i=0;i<n;i++){
19         for(j=0;j<n;j++){
20             arr1[i][j] = i;
21             arr2[i][j] = i;
22             ans[i][j] = 0;
23         }
24     }
25
26     start = clock();
27
28     #pragma omp parallel private(i,j,k) shared(arr1,arr2,ans)
29     {
30
31         #pragma omp for collapse(2) schedule(dynamic,1000)
32             for(i=0;i<n;i++){
33                 for(j=0;j<n;j++){
34                     trp[i][j] = arr2[j][i];
35                 }
36             }
37
38         #pragma omp for collapse(2) schedule(dynamic,5000) nowait
39             for(i=0;i<n;i++){
40                 for(j=0;j<n;j++){
41                     for(k=0;k<n;k++){
42                         //ans[i][j] += arr1[i][k]*arr2[k][j];
43                         ans[i][j] += arr1[i][k]*trp[i][k];
```

```
                    }
                }
            }
    }

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Sequential time taken = %lf\n",cpu_time_used);

    return 0;
}
```

3 Openacc

OpenACC is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems. Openacc stands for Open Accelerators. OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors. It is more or less similar to that of openmp except for some of the directives and clauses. Openacc allows the user to run the code either serially or multi-core CPU or the in the GPU. If some part of the code is to be accelerated, then it sends the necessary variables in the GPU and does the computation there and copies the results back.

3.1 Directives

Similar to openMP, openacc has a list of directives which are added to a block of code. The directives are placed after "*#pragma acc*" and before any clause.

Directives	Use
kernel	The compiler needs to decide whether that section can be parallelized or not.
parallel	The user decides whether that section can be parallelized or not.
loop	This directive tells the compiler that it is a parallel block of code.
data	This directive is used to transfer data to and fro the GPU in a structured way. It has mainly four clauses like copyin, copy, copyout, create and present.
enter data	This directive is used to transfer data to and fro the GPU but in a unstructured way. It has mainly four clauses like copyin, copy, copyout, create and present.
update	Updates the variable either in the GPU or the CPU as specified.
wait	Waits until the specified task is completed
async	It allows that block to run asynchronously.

Clauses	Use
private	Specifies that each thread should have a own copy of that variable.
shared	Specifies that the variable is shared among all other threads
reduction	Specifies that the private variable of each thread are to be reduced as specified.
independent	Forcing the compiler to parallelize the code even if the compiler feels it can't be.
sequence	The block executes the code in a sequential order.
copy	Copies the variable, in the GPU from CPU before the start of the block and in the CPU from GPU after the end of the block.
copyin	Only copies the variable from CPU to GPU.
copyout	Only copies the variable from GPU to CPU after the end of the block.
create	Creates a variable in the GPU of the specified memory.
present	Tells the compiler that the variable is already present in the GPU.
schedule	This clause is only for loop directive. It allows to divide the work in a static or dynamic way.

3.2 How to speed-up using openacc

In this sub-section, I will explain the basic four steps on how one can speed-up his code using openacc. The four basic steps are identify parallelism, parallelize using openaccc, optimize data locality and optimize loop.

3.2.1 Identify Parallel regions

The foremost step for a user is to identify the regions which can be parallelized. Then try adding "*#pragma acc kernels*" to that section and infer the information while compiling. In most of the cases, the compiler can say that complex loop carried dependency and hence cannot be parallelized.

3.2.2 Parallelize Loops

Once we have identified the parallel loops, we can add "*#pragma acc parallel loop*" to the parallel loop and then compile it with Minfo flag set to all which allows the user to see all the compiler information. If the compiler disagrees with the user and prevents parallelism, then if the user is completely sure that the loop can be parallized then he can add an *independent* clause after the loop directive which forces the compiler to parallelize the loop without actually caring about he risk involved.

3.2.3 Optimize Data Movement

After the successful compiling of the code, we will actually see a dip in the speed-up of the code. It becomes even more slower even after parallelizing it which doesn't make sense. The trick here is that, the compiler doesn't know when to transfer the data to and fro the GPU and hence it ends up doing it a lot more time than required which slows down the code. To handle that issue, we need to take control over that and tell the compiler when to transfer the data. It can be done in either structured way or in an unstructured way.

Structured Way - "*#pragma acc data {}* "
Unstructured Way - "*#pragma acc enter data*"

Both ways have clauses like copyin, copyout, copy, create but only unstructured way has delete clause. In a structured way, there is curly brackets which contains the data movement. And the data exits or deletes only at the end. While in unstructured way, we can send the data to GPU whenever we want and copy it back or delete it if that variable is no longer required.

After this step, we can see significant speed-up in the code. We can further increase it by following the fourth step.

3.2.4 Optimize Loop Performance

Here we do work in asynchronous manner "*#pragma acc async(1)*", i.e. pre-fetching the data required for the next loop. Then we have to add the wait clause to ensure that the asynchronous task has been completed. With this step, we can increase the speed-up but not to a much higher extent.

3.3 Example of Matrix multiplication

Listing 2: Matrix Multiplication

```
#include<openacc.h>
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

int n = 2000;
int arr1[2000][2000];
int arr2[2000][2000];
int ans[2000][2000];
int trp[2000][2000];
int main(int argc, char const *argv[])
{
    /* code */

    clock_t start, end;
    double cpu_time_used;
    int sum = 0;
    double diff,t;
    int i,j,k;

    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            arr1[i][j] = i;
            arr2[i][j] = i;
            ans[i][j] = 0;
        }
    }
    start = clock();
    #pragma acc data copyin(arr1,arr2,n) create(i,j,k,sum,trp) ↩
       copyout(ans)
    {
        #pragma acc parallel loop gang
            for(i=0;i<n;i++){
                #pragma acc loop vector
                    for(j=0;j<n;j++){
                        trp[i][j] = arr2[j][i];
                    }
            }

        #pragma acc parallel loop worker
```

```
            for(i=0;i<n;i++){
                #pragma acc loop gang
                    for(j=0;j<n;j++){
                        sum = 0;
                        #pragma acc loop gang reduction(+:sum)
                            for(k=0;k<n;k++){
                                sum += arr1[i][k]*trp[i][k];
                            }
                        ans[i][j] = sum;
                    }
            }

    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time taken by parallel = %lf\n",cpu_time_used);

    return 0;
}
```

4 Profiling Sequential, OpenMP and Openacc code

Dimension	Sequential Time	OpenMp Time	Openacc Time
100 X 100	0.01	0.009	0.746
250 X 250	0.085	0.041	0.751
500 X 500	0.666	0.312	0.76
750 X 750	2.121	1.105	0.774
1000 X 1000	3.575	2.682	0.866
1200 X 1200	5.193	3.584	0.926
1400 X 1400	8.256	5.853	0.975
1600 X 1600	14.272	12.182	1.107
1800 X 1800	17.876	12.682	1.233
2000 X 2000	42.64	20.66	1.388
3000 X 3000	153.092	70.96	2.46

5 Conversion into Openacc

There are nearly 8 to 9 for loops inside the main simulation loop. To increase the speed-up, we have to parallelize as many loops as possible. In each paragraph, the problems related to conversion will be discussed.

Listing 3: Finding Active Rays

```
1     for(int i3=0;i3<N;i3++){
2         if(status[i3]==0){
3             ii[nii]=i3;
4             nii++;
5         }
6     }
```

The problem faced here is if we parallelize this then there will be locks on nii variable and hence it will delay the process than required. The other alternative is we can remove the block and run all other for loops till N and in the beginning check whether the ray is active(status = 0) or not. I'm following the latter approach and have completely removed the block. This makes the code run even faster.

Listing 4: Random Number Loop

```
1 for(iii=0;iii<nii;iii++){
2     phi0[ii[iii]]=(1.0*rand()/RAND_MAX)*2*PI;
3     u[iii] = (1.0*rand()/RAND_MAX);
4 }
5 #pragma acc update device(u[:N],phi0[:N])
```

Listing 5: Pseudo Random Generation & its Use

```
1 #pragma acc routine seq
2     unsigned long rnd(long prev){
3         unsigned long a = 1103515245,m = 2147483648,c = 12345;
4         unsigned long nt = a*prev+c;
5         nt = nt%m;
6         return nt;
7     }
8
```

```
#pragma acc parallel loop gang private(prev,prev1)
    for(iii=0;iii<N;iii++){
        prev = m*r_u[iii];
        prev1 = m*r_phi[iii];

        prev = rnd(prev);
        u[iii] = (prev*1.0)/m;
        r_u[iii] = u[iii];

        prev1 = rnd(prev1);
        phi0[iii] = (prev1*1.0)/m;
        r_phi[iii] = phi0[iii];

    }
```

There is no rand() function in openacc and therefore this loop can't be accelerated. So in that case, we have to run the above two loops in CPU and then transfer the variables to GPU which costs a lot and slows down the overall process. The alternative is define a pseudo random generator function and then accelerate it. But in this process the initial seed must be different for each threads. Therefore, the threads must be limited. I tried the above mentioned approach and saw that the above approach alone consumes nearly 0.08 to 0.1s per loop. So, I tried defining a pseudo random generator in GPU and run it with a single thread. This process is much better as it takes only 0.01s per loop. Both the methods are listed in the above listing. Both the above methods are slow. Therefore the other alternate was to pass N seeds into GPU and use pseudo random generator in GPU. This makes the loop efficient and is now much faster.

Listing 6: tj Loop

```
#pragma acc parallel loop gang independent present(tj,P,Pinv,angle)
    for(int j2 = 0;j2<nj-1;j2++){
        #pragma acc loop vector
            for(int temp = 0;temp<N;temp++){
                //@@Change 7 to j2
                if(u[temp]>=P[j2][1] && status[temp] == 0){
                    tj[temp][j2] = splint((double *)P+j2*Nc+1,(↩
                       double *)angle+1,(double *)Pinv+j2*n,Nc-1,u[↩
                       temp]);
                }
                else tj[temp][j2] = 0.0;
```

```
10              }
11      }
```

This loop is parallelizable with no issues being faced here except that the all the shared variables must be present in the GPU.

Listing 7: theta0 Loop

```
1  #pragma acc parallel loop gang independent private(yp1,ypn,y2_0,↩
       y2_1,y2_2,y2_3,y2_4,y2_5,y2_6,y2_7) present(tj,sdepth)
2       for(int i2=0;i2<N;i2++){
3           if(status[i2] == 0){
4               yp1 = (tj[i2][1]-tj[i2][0])/(sdepth[1]-sdepth[0]);
5               ypn = (tj[i2][nj-2]-tj[i2][nj-3])/(sdepth[nj-2]-sdepth[↩
                    nj-3]);
6               spline1(sdepth,tj[i2][0],tj[i2][1],tj[i2][2],tj[i2][3],↩
                    tj[i2][4],tj[i2][5],tj[i2][6],tj[i2][7],nj-1,yp1,ypn↩
                    ,&y2_0,&y2_1,&y2_2,&y2_3,&y2_4,&y2_5,&y2_6,&y2_7);
7
8               theta0[i2]=splint1(sdepth,tj[i2][0],tj[i2][1],tj[i2↩
                    ][2],tj[i2][3],tj[i2][4],tj[i2][5],tj[i2][6],tj[i2↩
                    ][7],y2_0,y2_1,y2_2,y2_3,y2_4,y2_5,y2_6,y2_7,nj-1,↩
                    pos[2][i2]);
9           }
10      }
```

This loop is also parallelizable but the private variables consume a lot of memory and hence segmentation fault occurs. The private variable of each thread are i2,yp1,ypn,j3,ydep[:nj-1],sdep2[:nj-1]. Hence we can run it in GPU but with constraints on number of threads. This process shockingly consumes 8s. Hence to parallelize this, I wrote different versions of spline and spline which will reduce the number of private variables per thread. The alternate versions of spline and splint takes 8 variables instead of an array and hence the for loops of the function are opened.Then the loop was paralleled which significantly reduces the time taken to nearly 0.06s.

Listing 8: Theta0vals Loop

```
1  #pragma acc loop seq
2       for(int iii=0;iii<N;iii++){
3           if(theta0[iii]>0.08){
```

```
4               iit[count]=iii;
5               count++;
6           }
7           //if(theta0[iii]==0) temp++;
8       }
9   printf("len(iit)=%d\n",count);
10
11  //#pragma acc update self(count)
12  if(count>1000) l1=1000;
13  else l1=count;
14
15  //#pragma acc update device(l1,l)
16  //#pragma acc parallel loop
17      for(count = l;count<l+l1;count++){
18          theta0vals[count][1]=theta0[iit[count-l]];
19          theta0vals[count][0]=pos[2][iit[count-l]];
20      }
```

In this loop, the same problem occurs as in the first loop and hence to initialize the iit variable it must be done in CPU or with one thread in GPU. This is again a barrier loop which restricts the speed-up. The second for loop in this listing can be parallelized. The problem of running the first loop in CPU again puts barrier on speed-up due to memory transfers. But this listing of code is not used anywhere after the main simulation loop. Hence I have commented this listing in the code.

Listing 9: Updation Loop

```
1   #pragma acc parallel loop gang independent private(sx,sy,sz,phi,↩
       theta1,cosphi,costheta,sinphi,sintheta) present(status[:N],phi0↩
       [:N],flx[:3][:N],pos[:3][:N],x_att[:n_att],y_att[:n_att],y2_att↩
       [:n_att])
2       for(iii=0;iii<N;iii++){
3           if (status[iii] == 0){
4
5               sx= cos(phi0[iii])*sin(theta0[iii]);
6               sy= sin(phi0[iii])*sin(theta0[iii]);
7               sz= cos(theta0[iii]);
8               phi=atan2(flx[1][iii],flx[0][iii]);
9               theta1=atan2(sqrt(pow(flx[0][iii],2)+pow(flx[1][iii],2)↩
                   ),flx[2][iii]);
10              cosphi=cos(phi);sinphi=sin(phi);
```

```
11              costheta=cos(theta1);sintheta=sin(theta1);
12
13              flx[0][iii]= (cosphi*costheta*sx) + (-sinphi*sy) + ↩
                   cosphi*sintheta*sz;
14              flx[1][iii]= (sinphi*costheta*sx) + (cosphi*sy) + ↩
                   sinphi*sintheta*sz;
15              flx[2][iii]= (-sintheta*sx) + costheta*sz;
16
17              //intensity[ii[iii]]=intensity[ii[iii]]*exp(-splint((↩
                   double *)Att+1,(double *)Att+13*i+1,y2_att,n_att,pos↩
                   [2][ii[iii]]));
18              intensity[iii]=intensity[iii]*exp(-splint(x_att,y_att,↩
                   y2_att,n_att,pos[2][iii]));
19
20              pos[0][iii] = pos[0][iii]+flx[0][iii];
21              pos[1][iii] = pos[1][iii]+flx[1][iii];
22              pos[2][iii] = pos[2][iii]+flx[2][iii];
23              if(pos[2][iii] < 0) pos[2][iii] = pos[2][iii]*(-1);
24          }
25
26      }
```

No issues were faced in accelerating this loop as the private variables per thread is less and it is independent of other iterations. The only issue is there are lot of shared variables but that is not of any concern as the GPU has 4Gb of shared memory.

Listing 10: Status Updation

```
1  #pragma acc parallel loop gang private(t2)
2      for(int i2=0;i2<N;i2++){
3          if(status[i2] == 0){
4              t2 = sqrt(pos[0][i2]*pos[0][i2]+pos[1][i2]*pos[1][i2]+(↩
                   pos[2][i2]-z0)*(pos[2][i2]-z0));
5              if(t2<=s){
6                  status[i2] = i;
7              }
8          }
9      }
10 #pragma acc update self(status[:N]) async(i) //Depending on the ↩
       first Loop
```

This loop is parallelized but depending on the how are we handling the first loop i.e. initializing ii variables, we need to update the status array in CPU in an asynchronous way to save time. If we run the first loop in GPU or remove the loop from the picture, then there is no need to update the status variable in CPU and it will save time. We need all the variables in the GPU for speed-up, hence I have eliminated ii array.

Listing 11: traj and beam2 Updation

```
#pragma acc parallel loop gang independent collapse(2) present(pos,traj)
    for(int tt = 0;tt<3;tt++){
        for(int tp = 0;tp<ntraj;tp++){
            traj[tt][tp][i]=pos[tt][tp];
        }
    }

#pragma acc parallel loop independent present(pos,beam2)
    for(int tt = 0;tt<N;tt++){
        if((pos[2][tt] > z2) && (beam2[tt][0]<0)){
            beam2[tt][0]=sqrt(pos[0][tt]*pos[0][tt]+pos[1][tt]*pos[1][tt]);
            beam2[tt][1]=atan2(pos[0][tt],pos[1][tt]);
        }
    }
```

This are independent loops and are paralleled by adding the accelerator directive.

6 Analyzing Performance in GPU

Copying all the variables from the CPU to GPU takes nearly 120 to 175s and similarly copying all the results back from the GPU takes 75 to 100s. Initially, I was running three loops(First loop, random number loop and theta0 loop) in CPU and copying memory to and fro the GPU and it used to take nearly 11 to 15s per loop which is nearly 3 times slower than the python code. I checked the GPU utilisation and it was only 35%. Then I tried the first loop and theta0 loop in GPU in sequential order. Hence limiting the memory transfer to random generator loop. But I didn't get much of an improvement due to sequential run of theta0 loop. And hence I modified splint and spline calls which paralleled the theta0 loop. Next issue was to generate random number in GPU. And as openacc doesn't support in-built rand() function, the only solution was to write a pseudo

random generator function in openacc. But giving different seeds to different threads was difficult, I am running this loop in only one thread and it takes nearly 0.009 to 0.01s per loop and hence consumes 43% of the time. The other alternative is I should pass a different seed to each thread. Therefore I'm now passing N seeds to GPU and hence this loop is also paralleled and this loop now runs much faster i.e. it takes only 0.001s.

6.1 Profiling the Simulation Code in Openacc

The below table depicts the time taken by GPU in seconds to run a loop of the simulation code. The openacc time includes the copyin (nearly 80s) and copyout (nearly 15s) time. The copyin and copyout depends on the number of simulations as the size of the variables changes.

Loop	Openacc
100	95.2
200	96.37
500	99.92
1000	105.65
2000	120.3
5000	162.33
10002	240

6.2 Profiling Comparison between Python, C, OpenMp and Openacc

The below table depicts the time taken by each of the architecture in seconds to run a loop of the simulation code. The openacc time includes the copyin (nearly 80s) and copyout (nearly 15s) time.

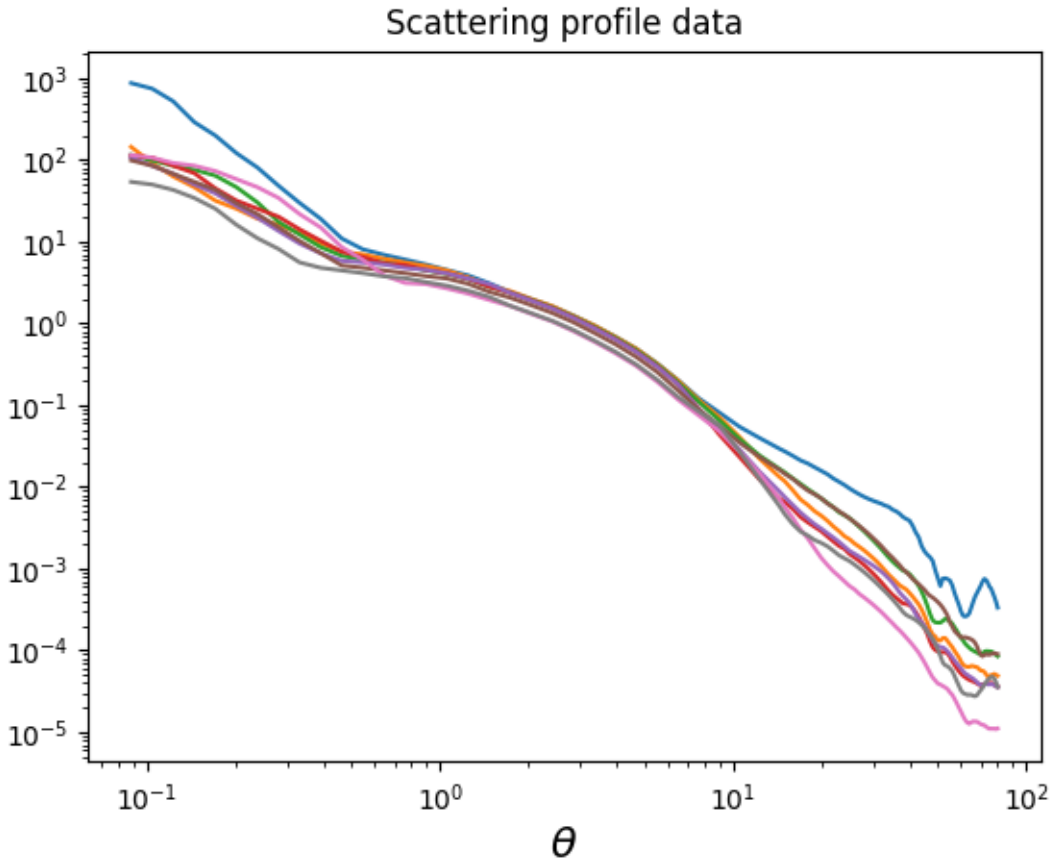
Loop	Python	C	OpenMp	Openacc
1	4	0.15	0.08	0.013
100	382	14.3	7.01	95.2
10002	-	1291.4	534.1	240

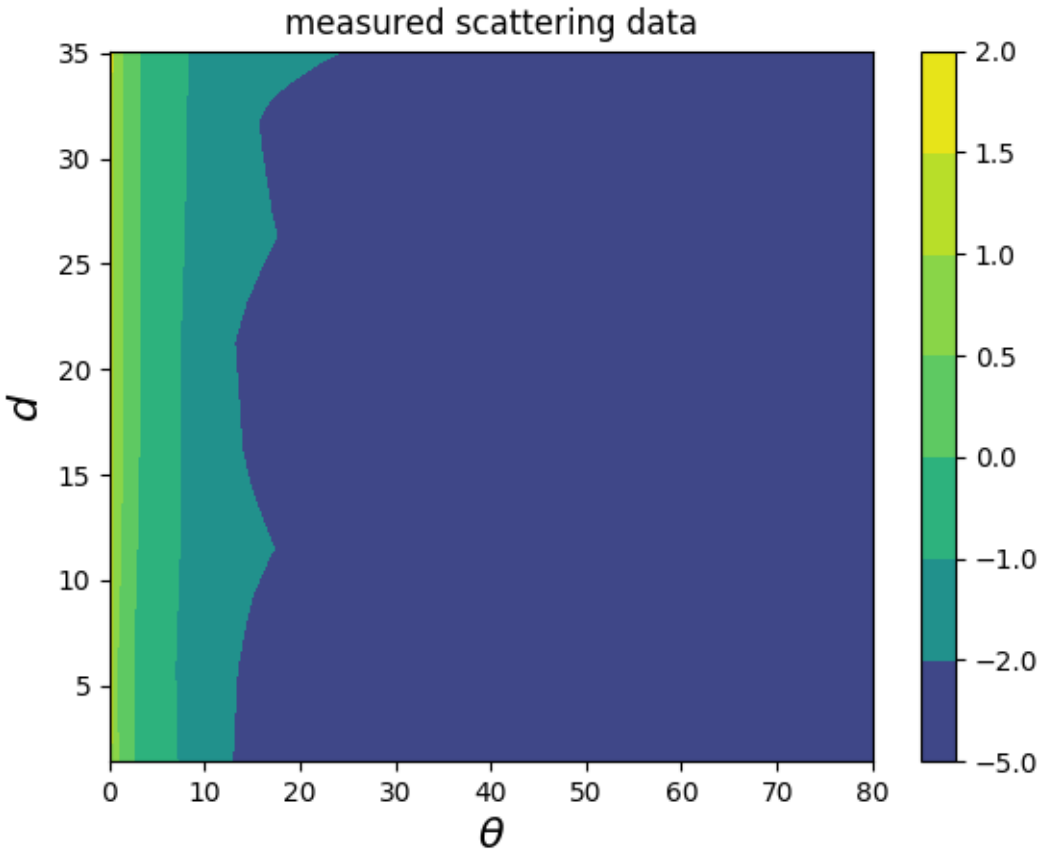
7 Plots

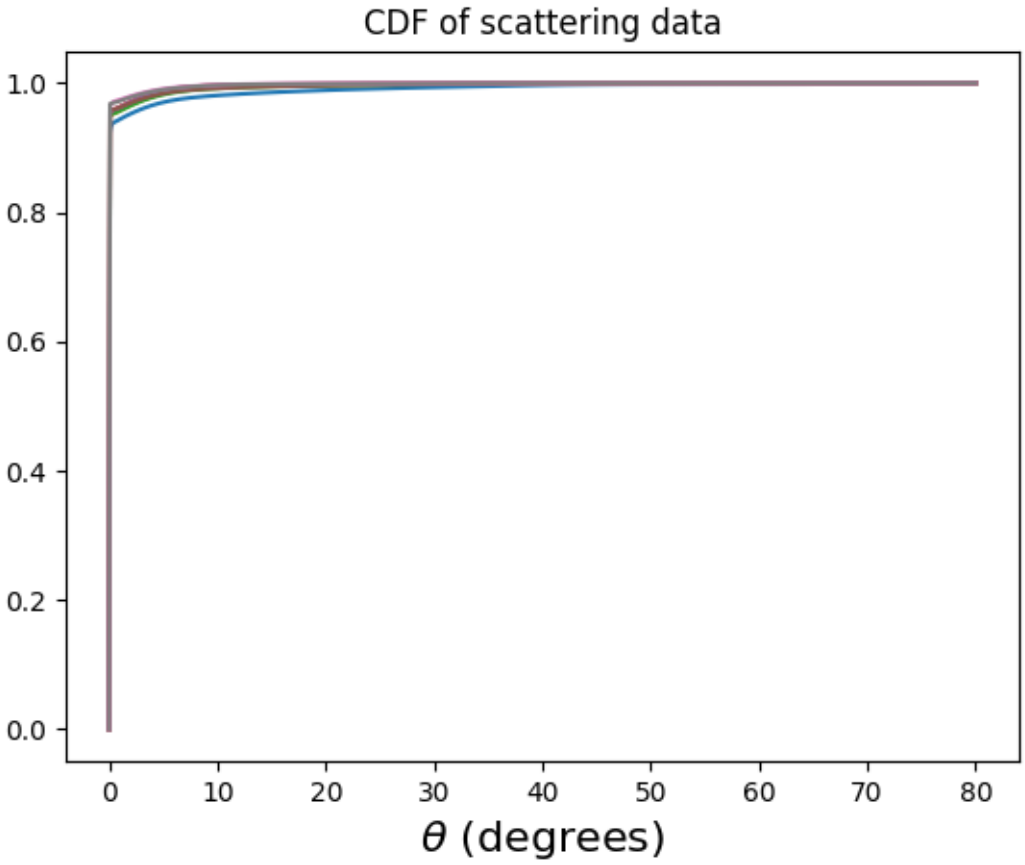
Here in this section, I will attach all the plots. This section is again divided into four sub parts. The first part contains three basic plots are plotted before the main simulation

loop and are same for all the cases. Remaining three part contains the output plots with some variation in the code.

7.1 Basic Plots



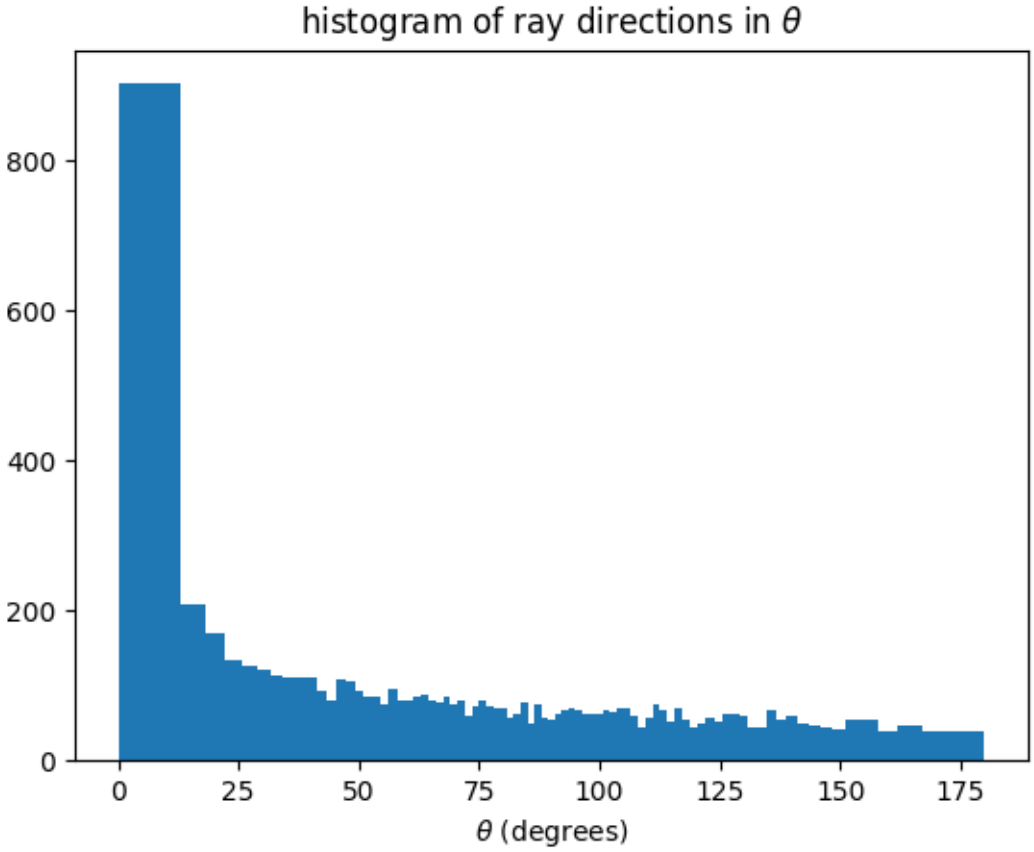


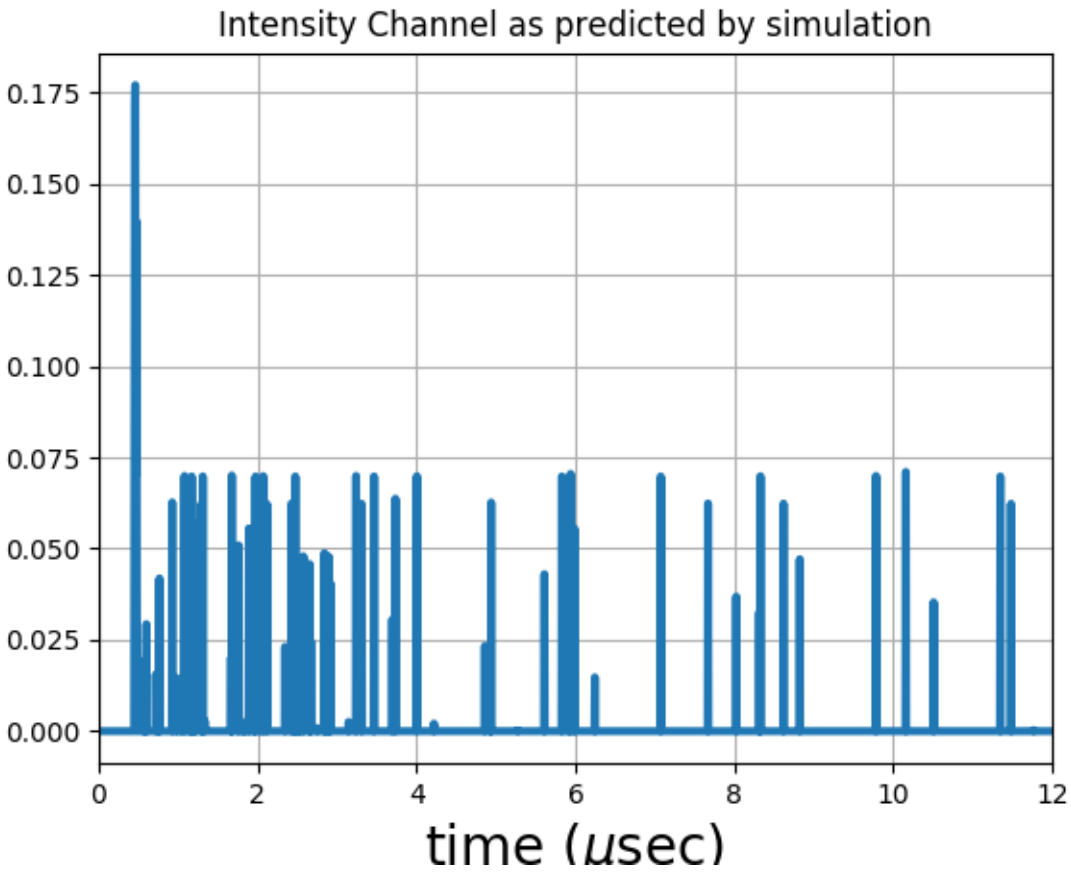


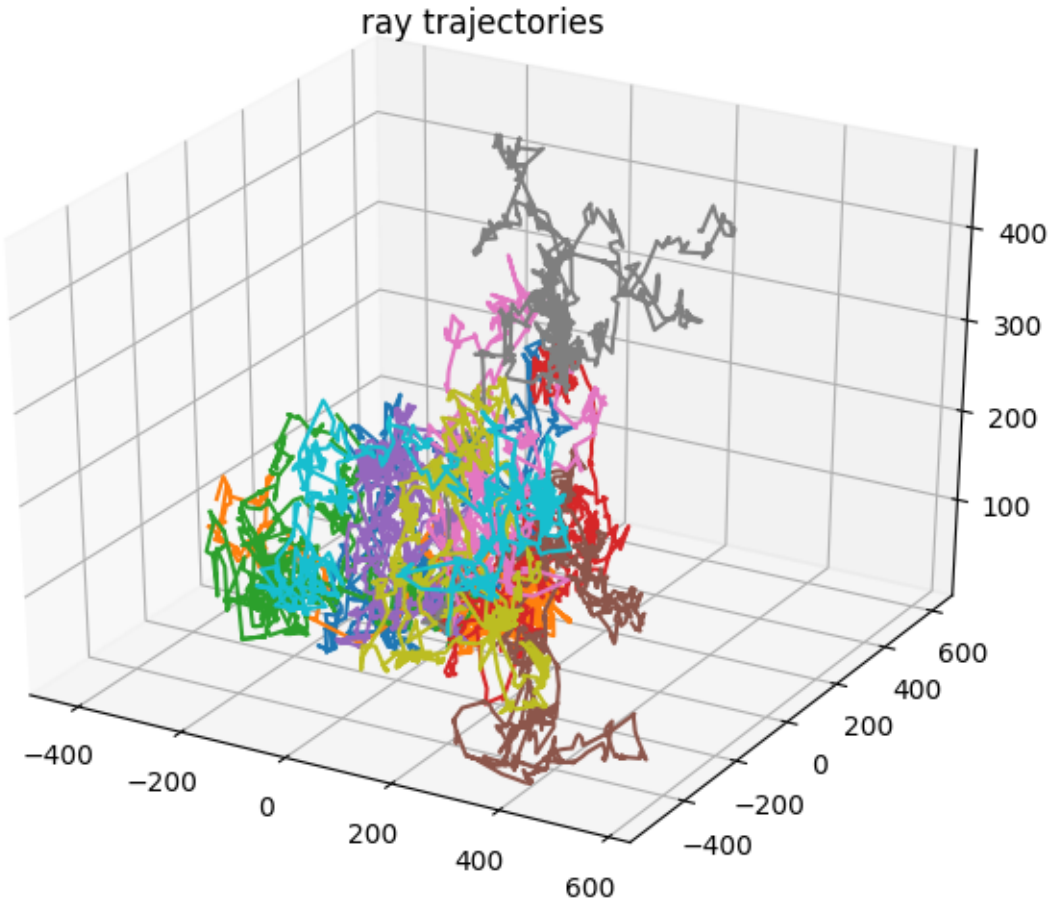
7.1.1 Initial Conditions

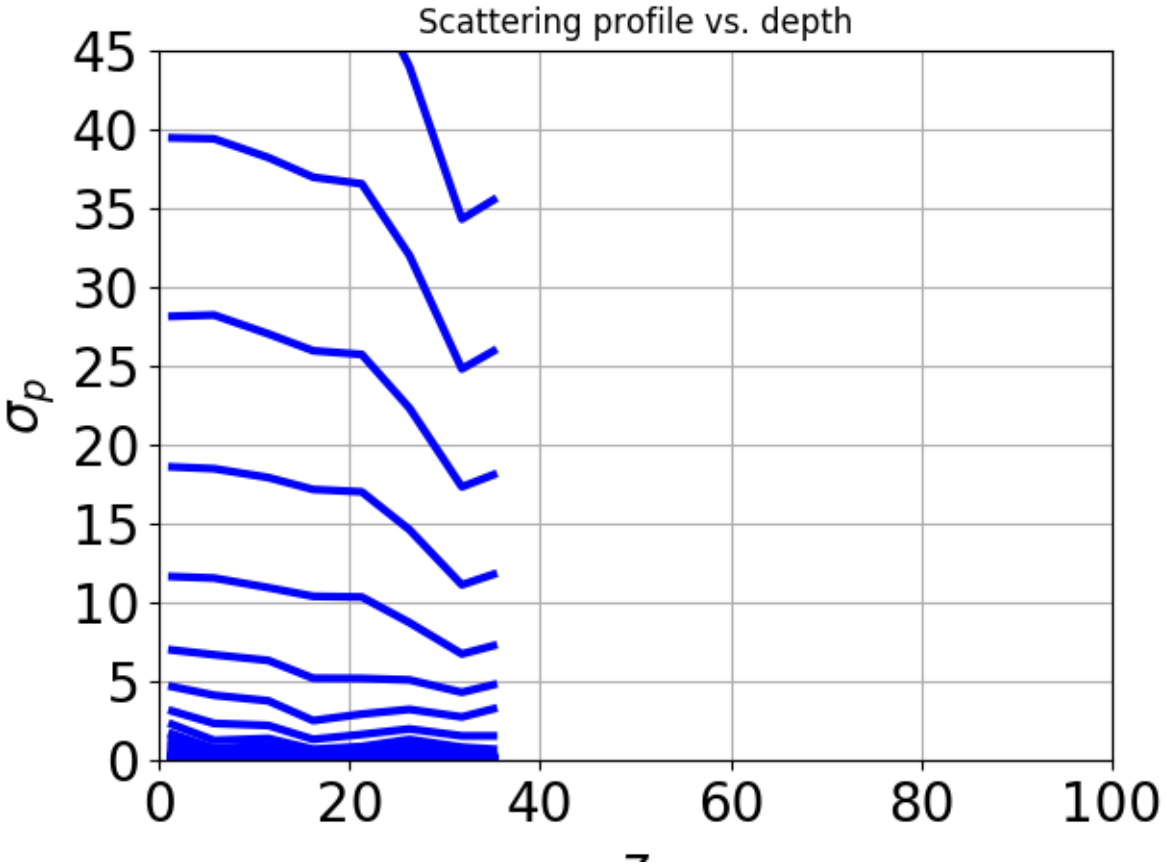
All the sections while plotting are run for 10002 iterations with 100000 rays i.e.
N = 100000
nt = 10002

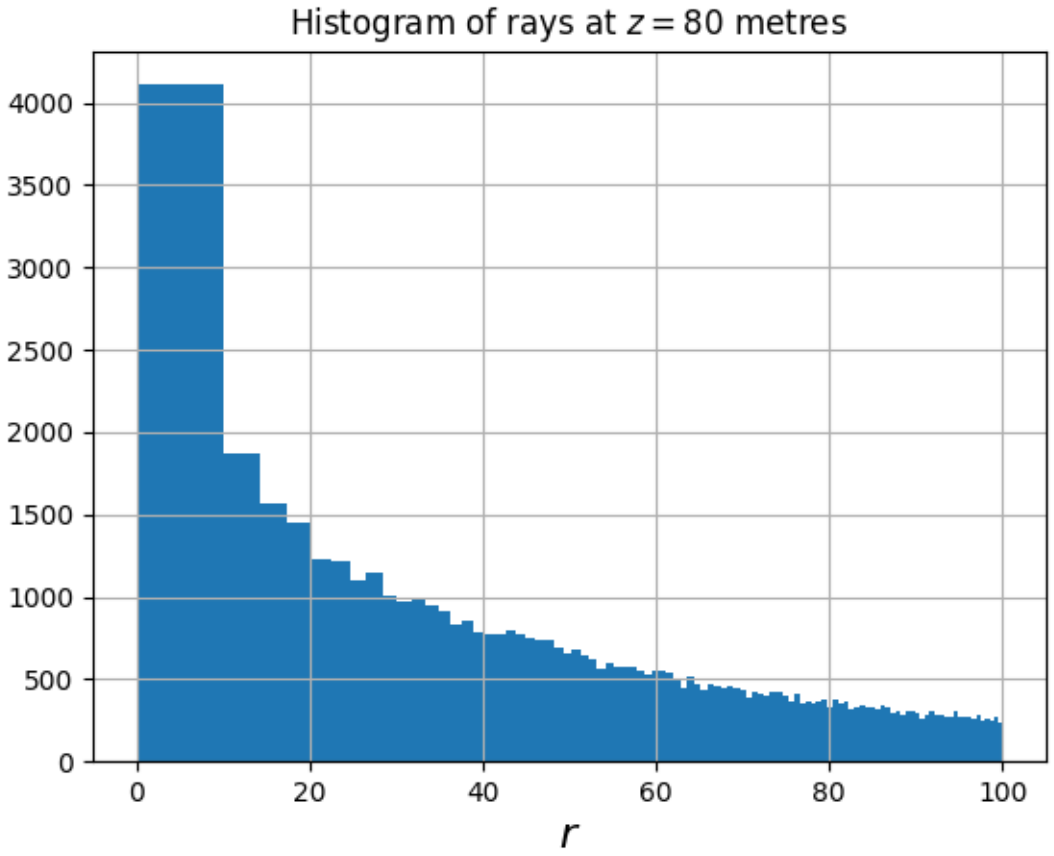
7.2 Plots without Constant Extrapolation






ray trajectories
400
300
200
100
600
400
200
0
−200
−400
−400
−200
0
200
400
600

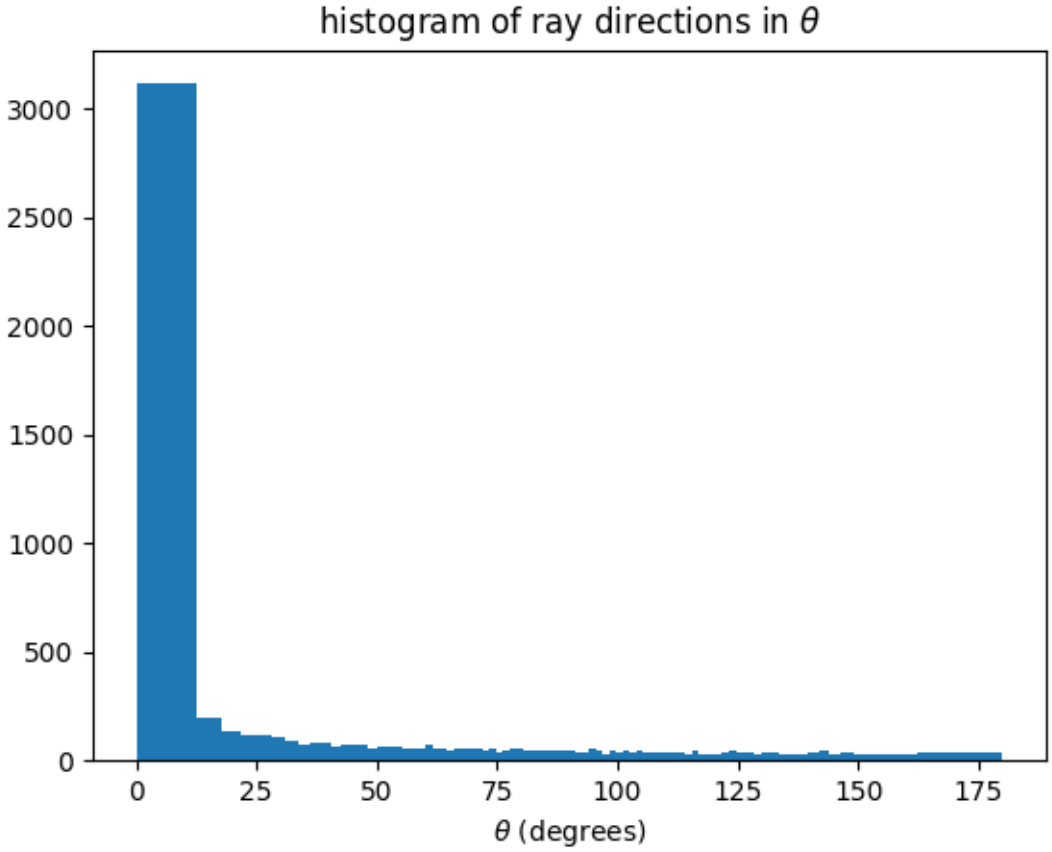

Scattering profile vs. depth
45
40
35
30
25
20
15
10
5
0
σ_p
0
20
40
60
80
100

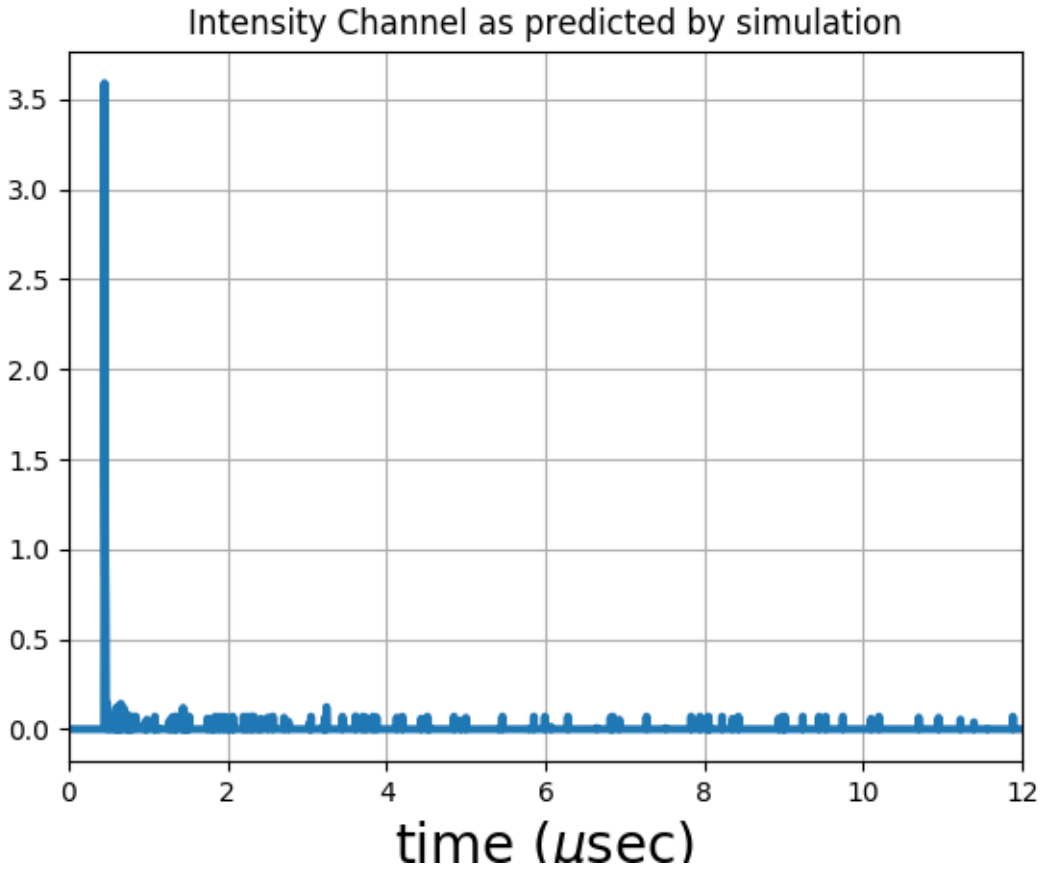


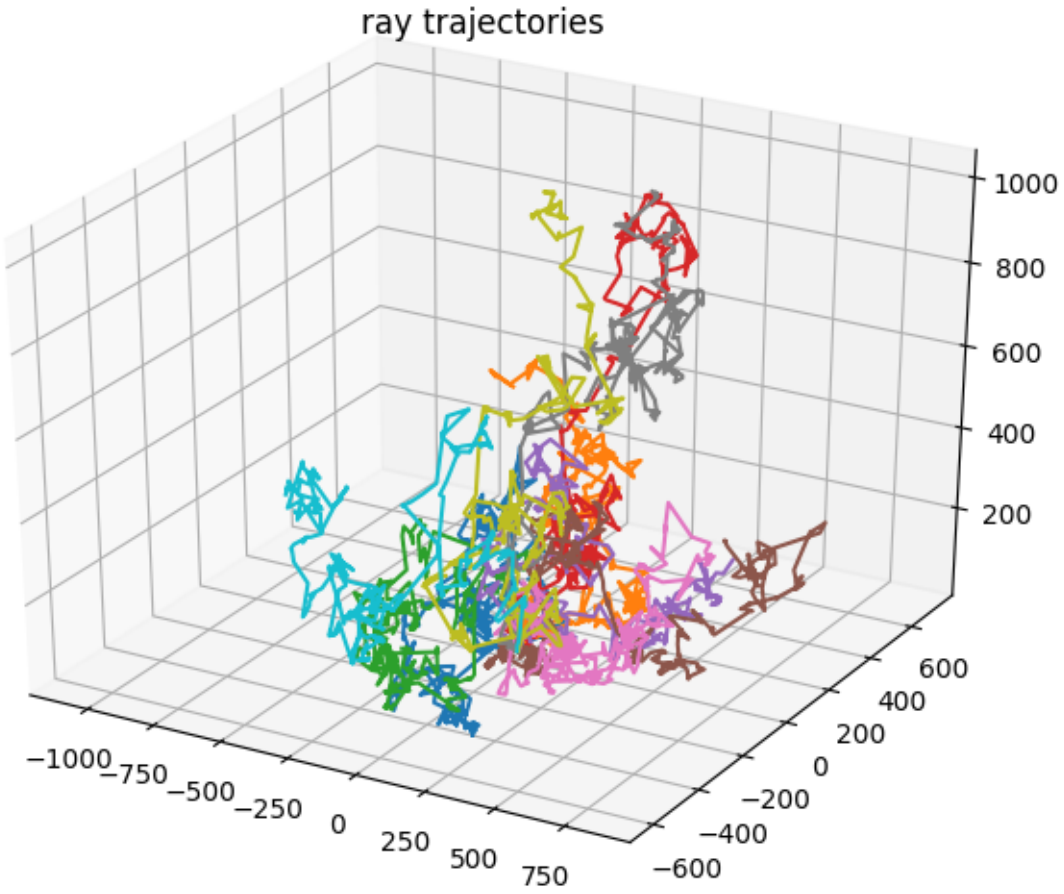
7.2.1 Output:

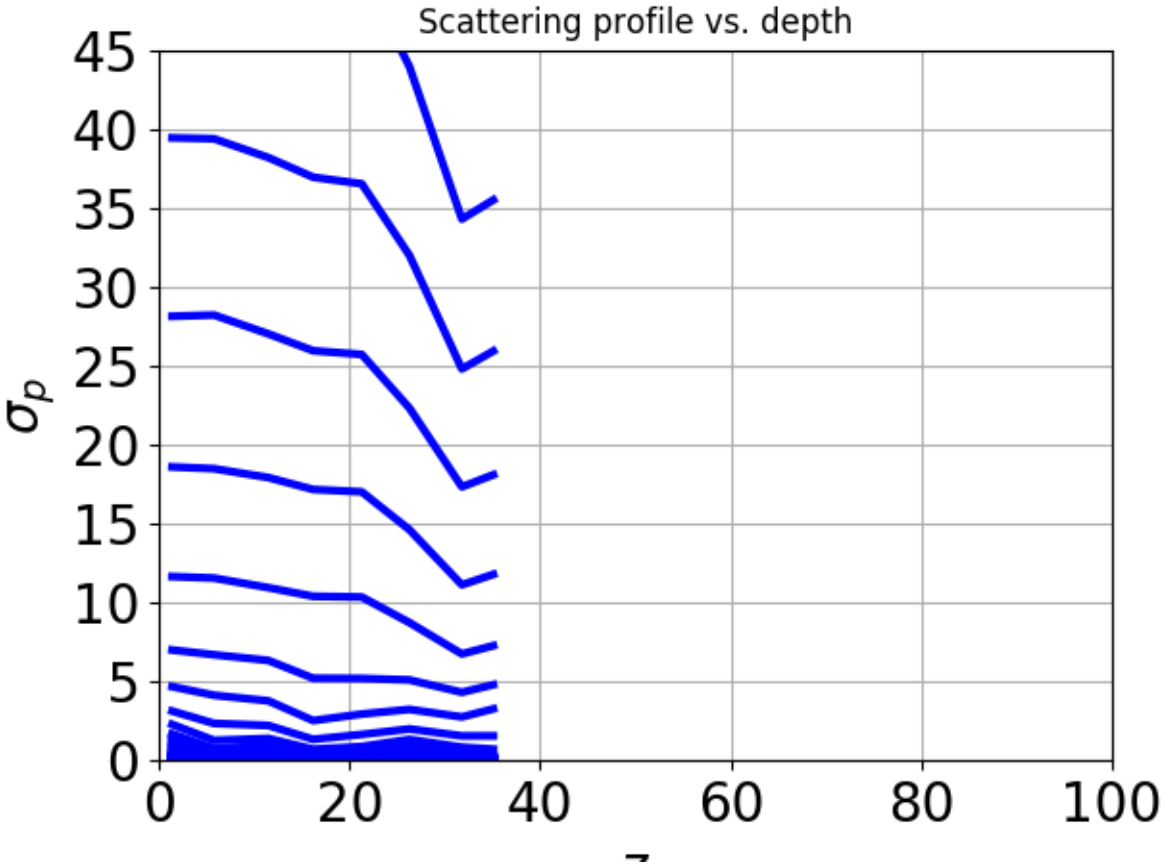
The output of the final code is:
6789 rays (out of 100000) reached the submarine
Average time to reach submarine=1602.29 time steps
Stdeviation of time to reach submarine=2172.08 time steps

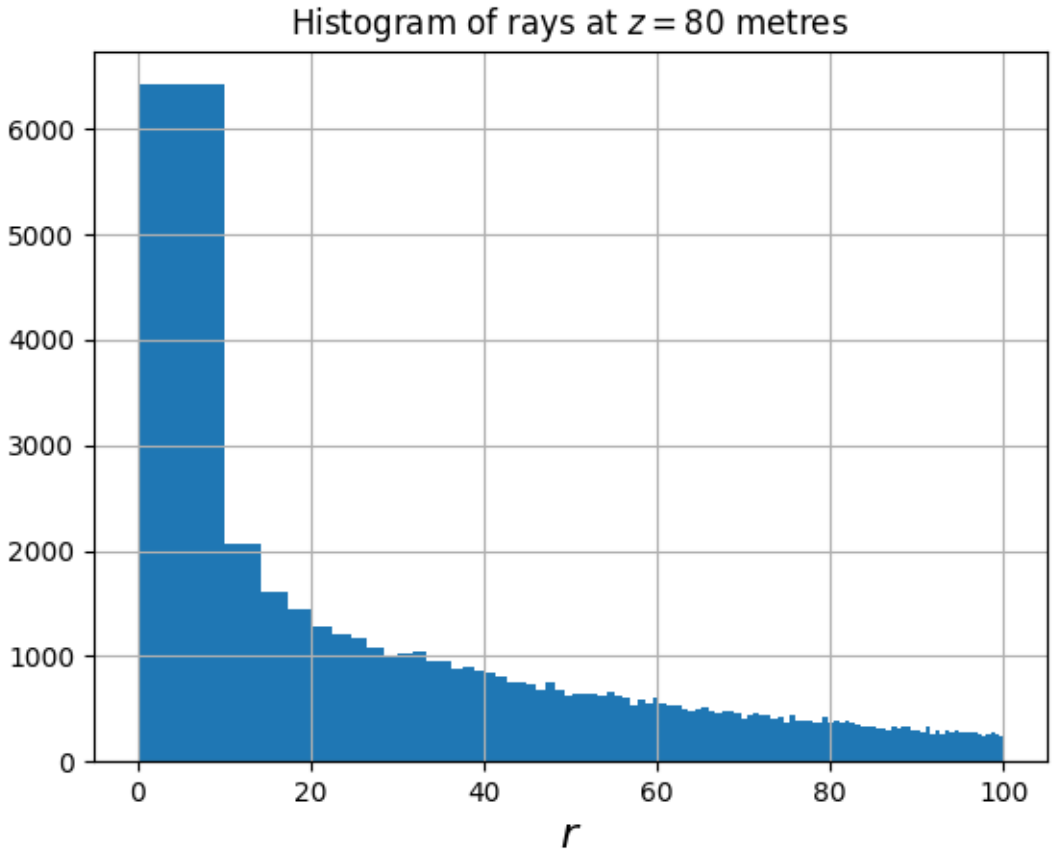
7.3 Plots of Constant Extrapolation






ray trajectories
1000
800
600
400
200
600
400
200
0
−200
−400
−600
−1000
−750
−500
−250
0
250
500
750


Scattering profile vs. depth
45
40
35
30
25
20
15
10
5
0
σ_p
0
20
40
60
80
100



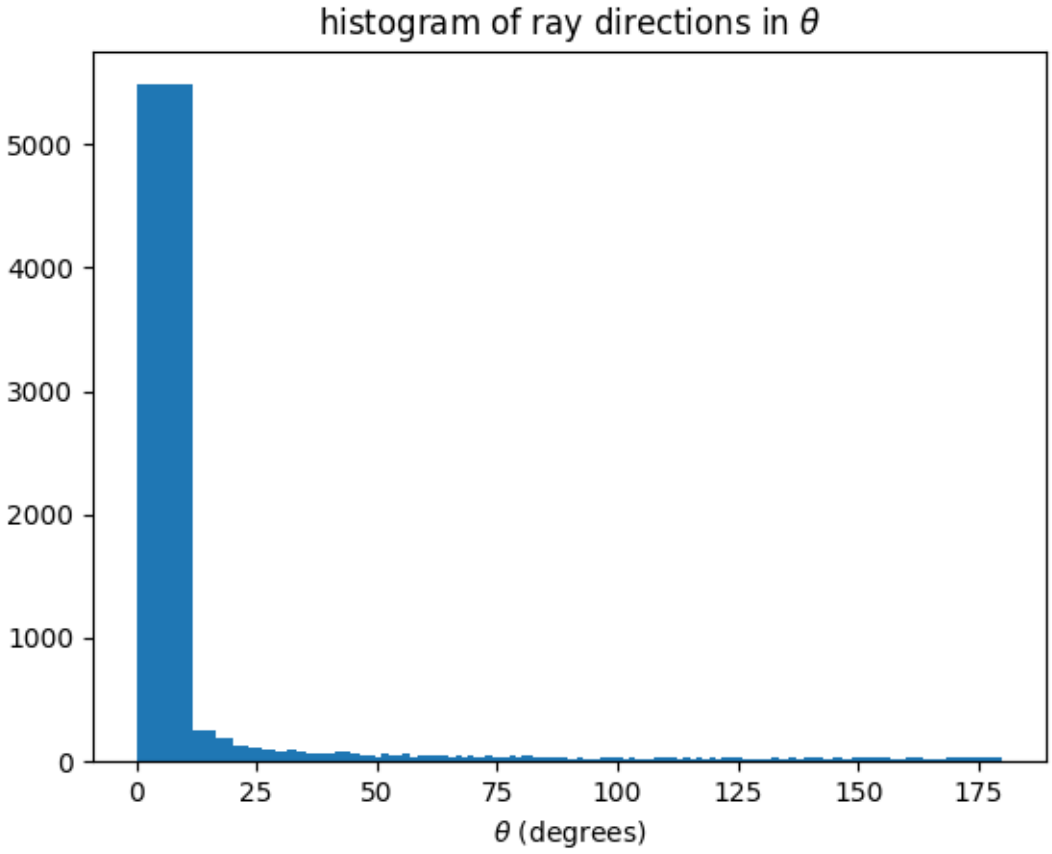
7.3.1 Conditions:

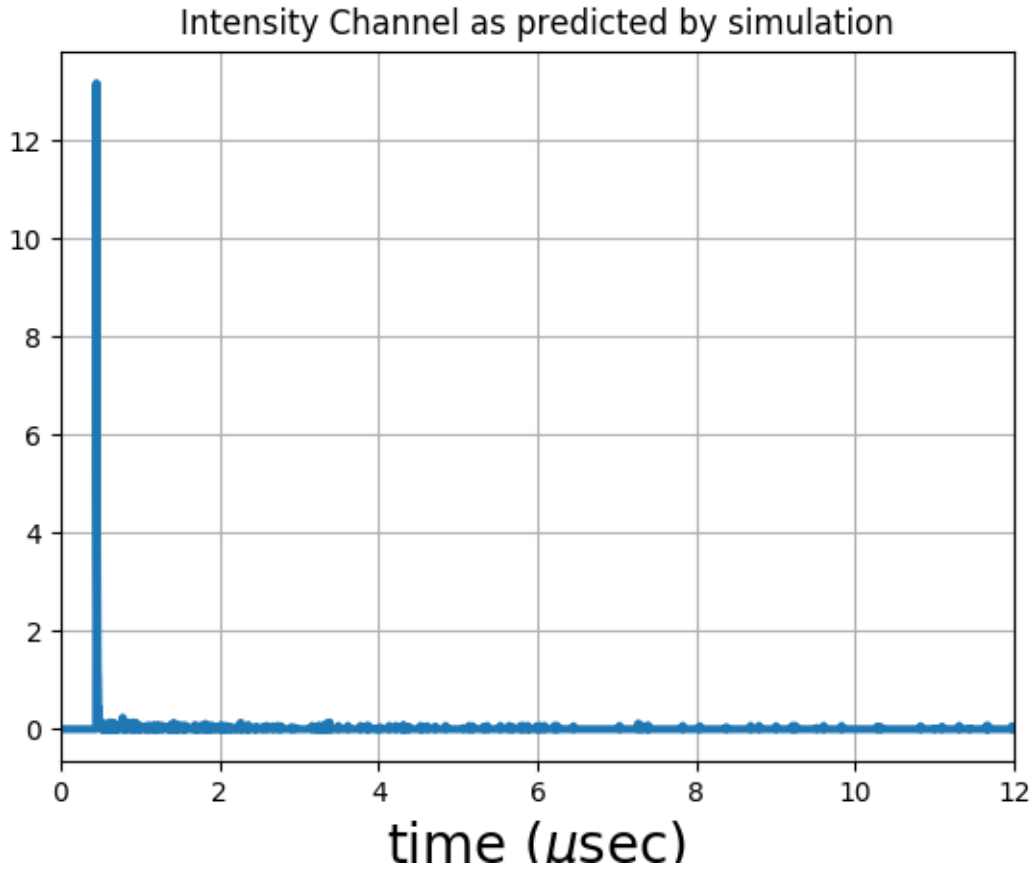
The splint function is modified as whenever its extrapolating, it gives a constant value of the first or the last coordinate depending on which side its extrapolating.

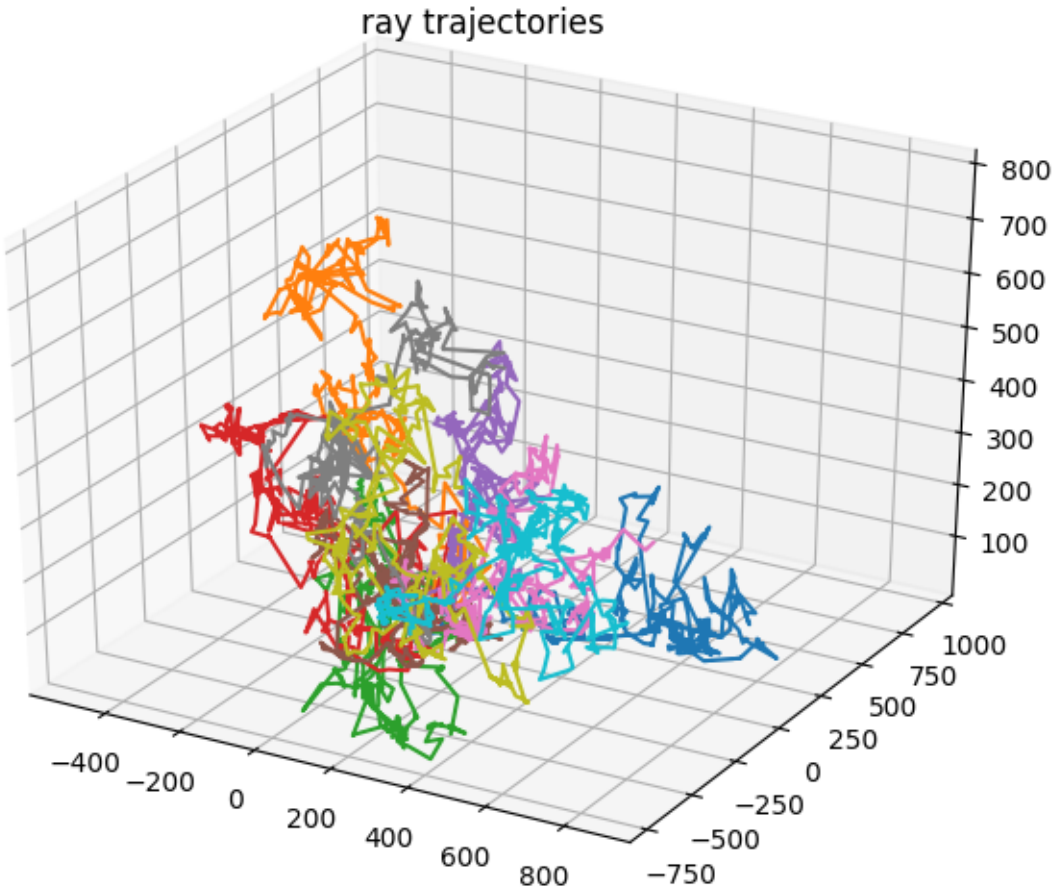
7.3.2 Output:

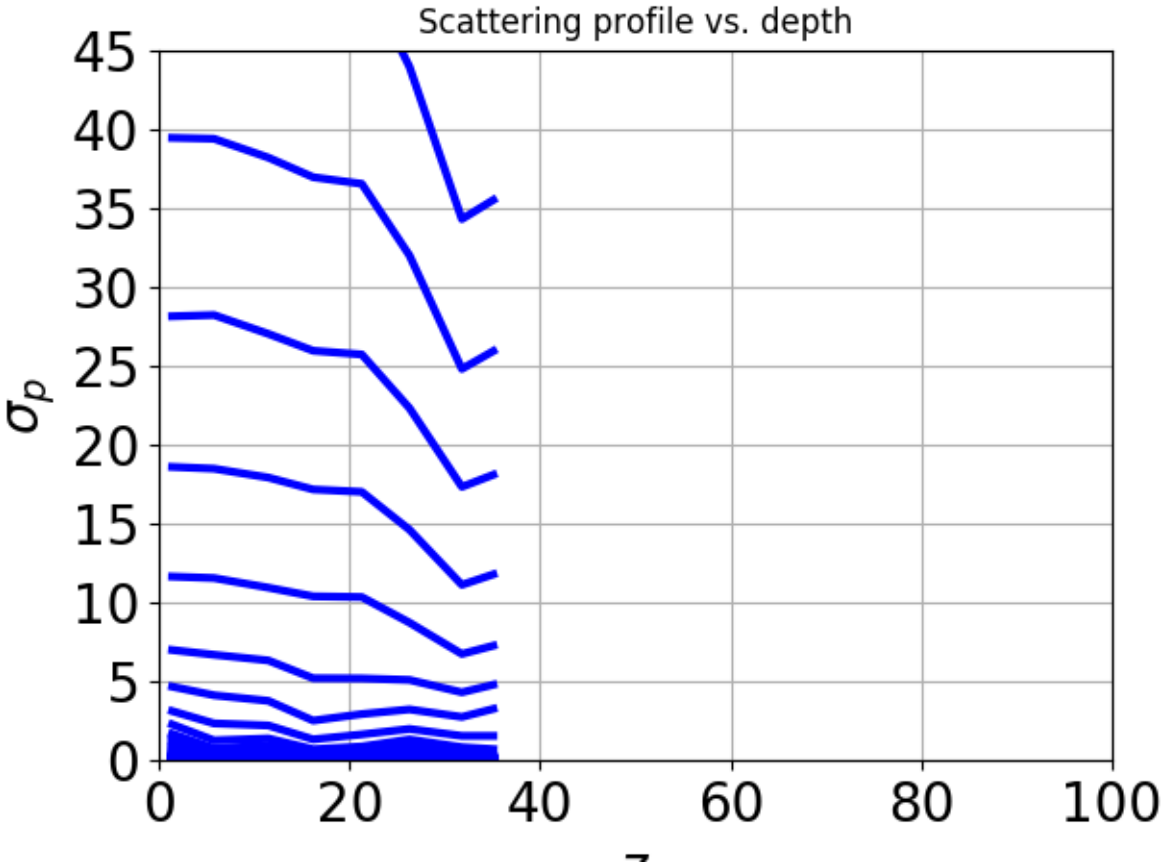
The output of the final code is:
7373 rays (out of 100000) reached the submarine
Average time to reach submarine=870.28 time steps
Stdeviation of time to reach submarine=1628.35 time steps

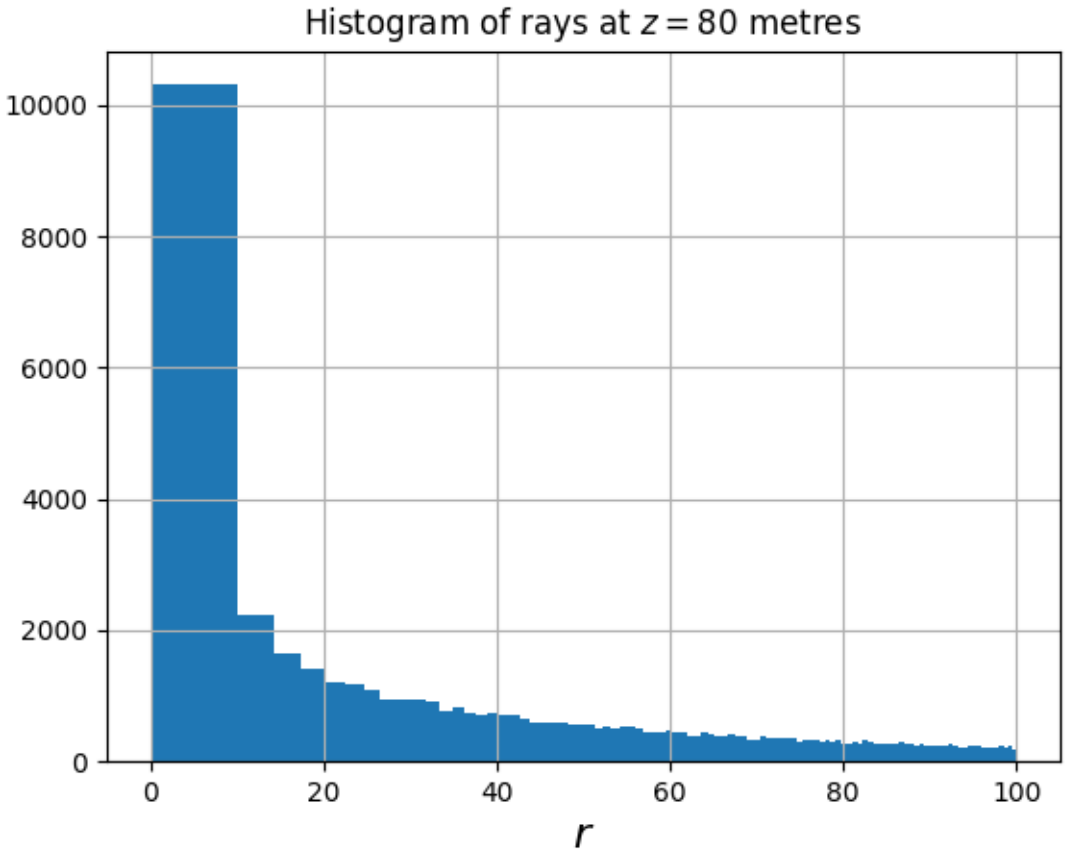
7.4 Plots with Index Constant






ray trajectories
800
700
600
500
400
300
200
100
1000
750
500
250
0
−250
−500
−750
−400
−200
0
200
400
600
800


Scattering profile vs. depth
σ_p
45
40
35
30
25
20
15
10
5
0
0
20
40
60
80
100



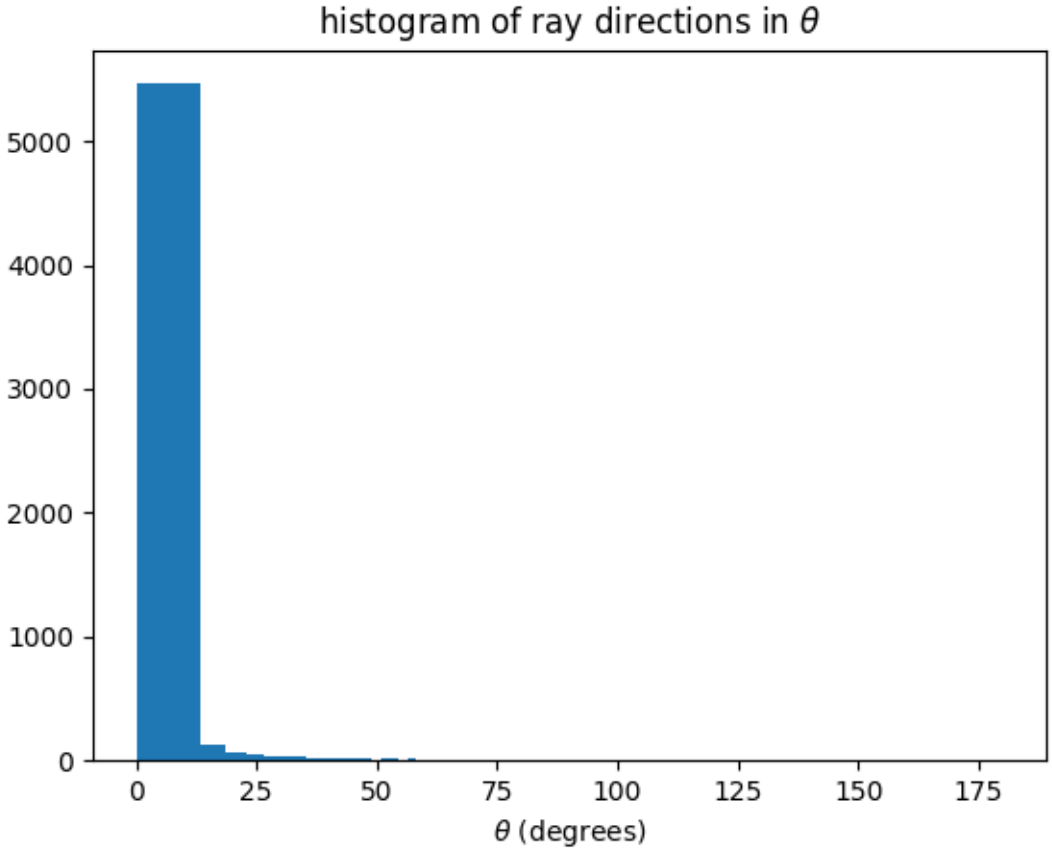
7.4.1 Conditions:

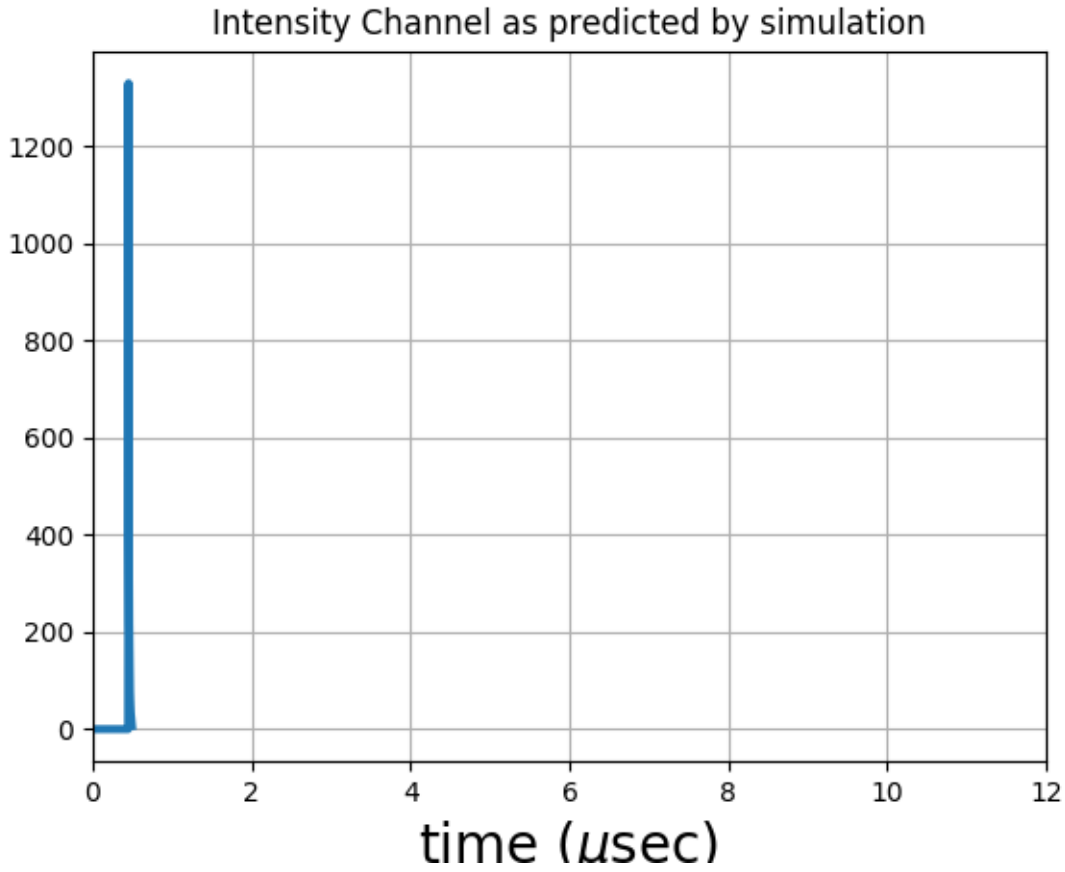
As we saw the plot of tj array in C and Python above, there was differences in the values but the value of the last point is same. And the value of a row in tj, in python, is same therefore I assigned all the values of tj to the values of its last column. Hence I got similar results as in the case of Python.

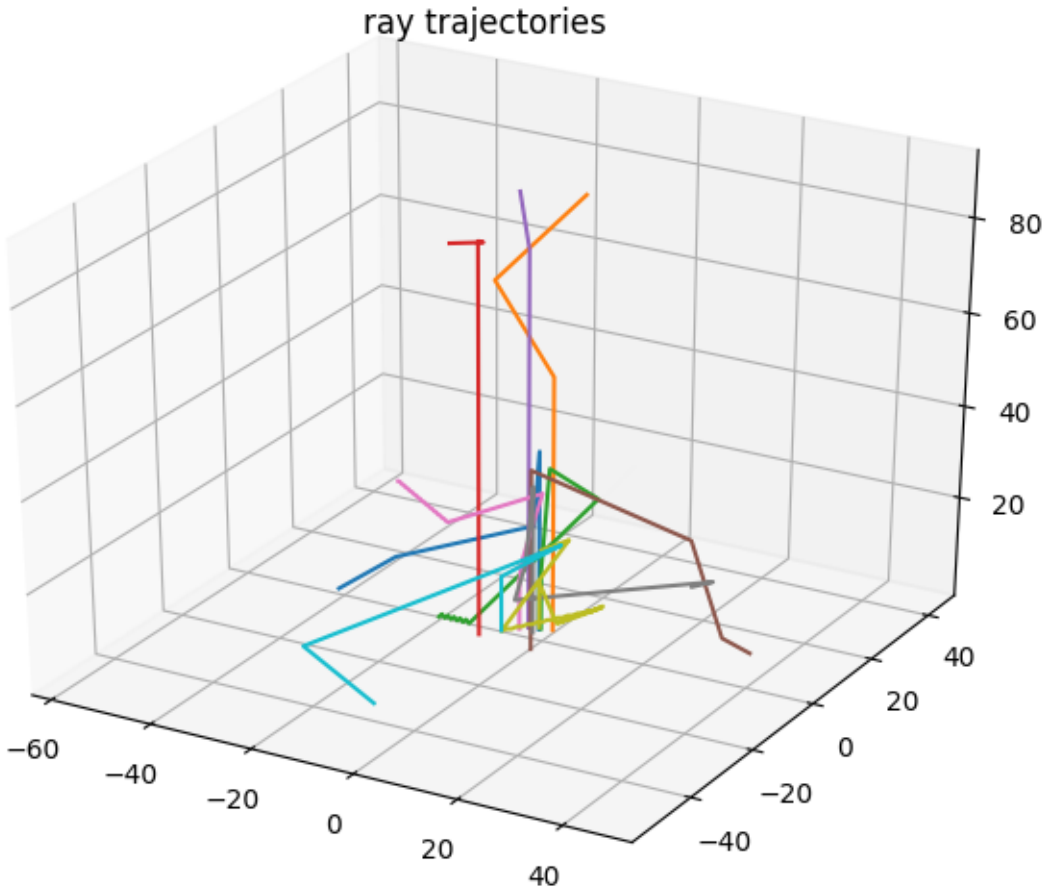
7.4.2 Output:

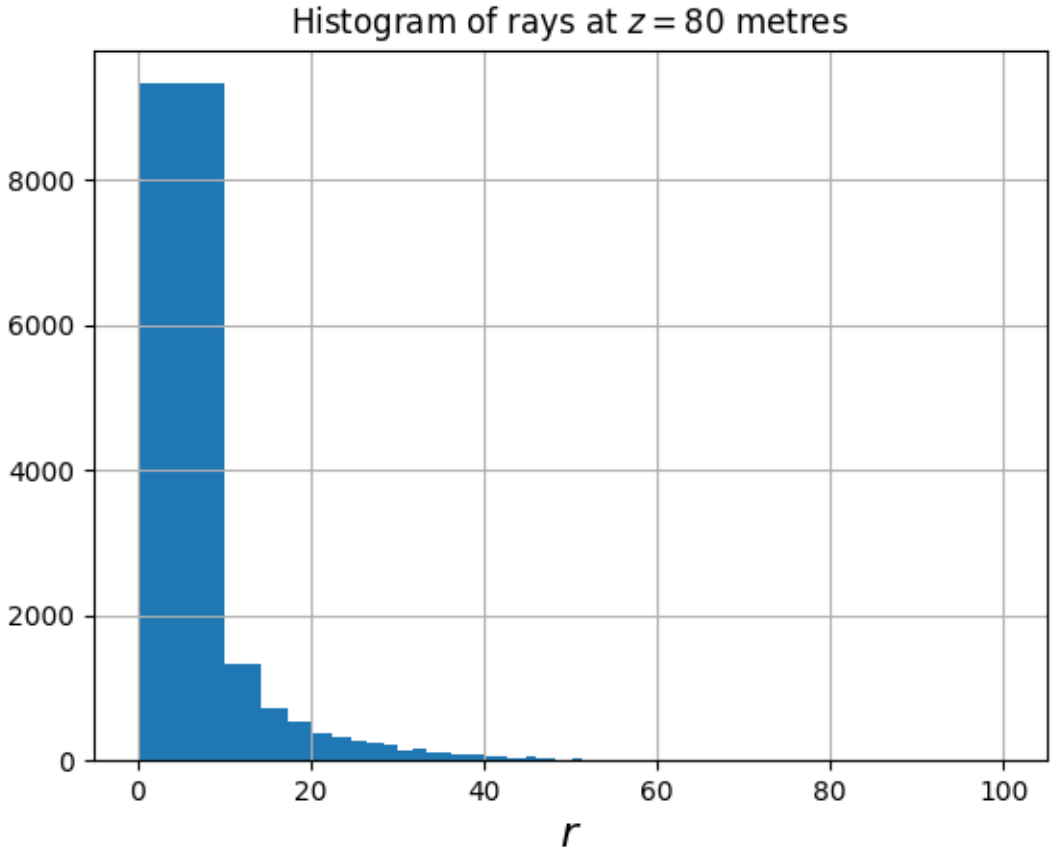
The output of the final code is:
9741 rays (out of 100000) reached the submarine
Average time to reach submarine=576.08 time steps
Stdeviation of time to reach submarine=1342.23 time steps

7.5 Plots from Python






ray trajectories
80
60
40
20
−60
−40
−20
0
20
40
40
20
0
−20
−40


Histogram of rays at $z = 80$ metres
8000
6000
4000
2000
0
0
20
40
60
80
100
r

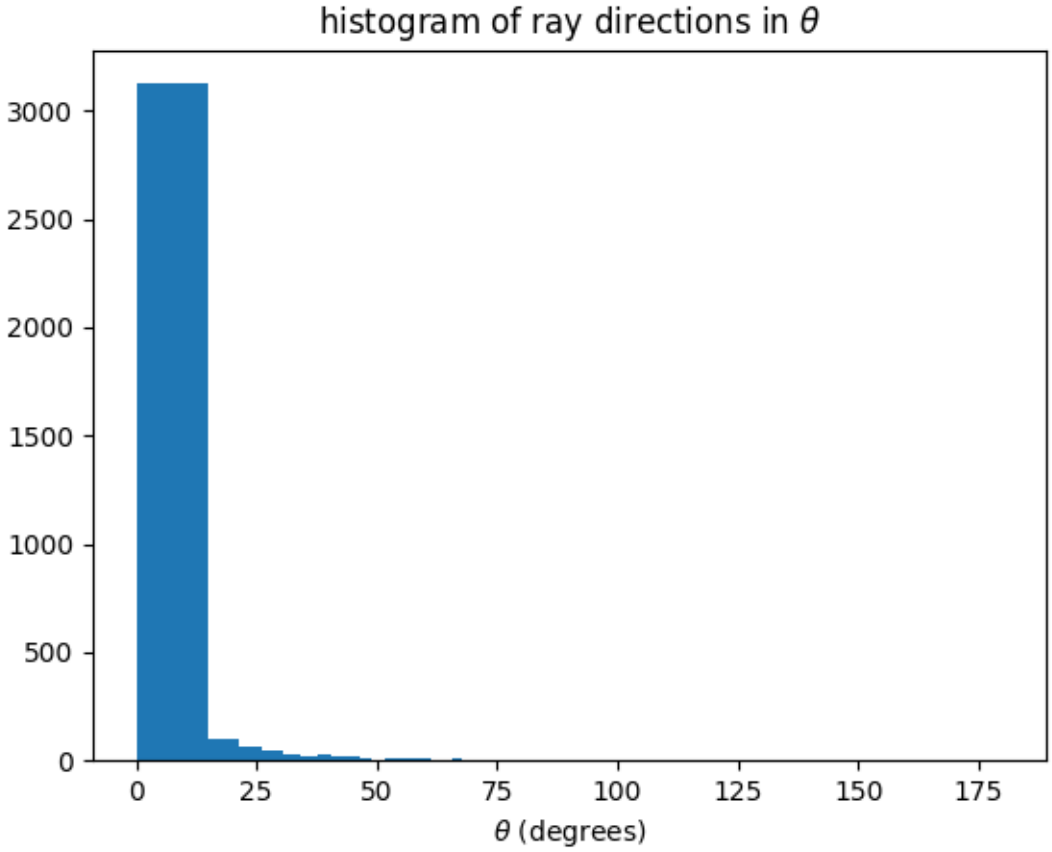
7.5.1 Conditions:

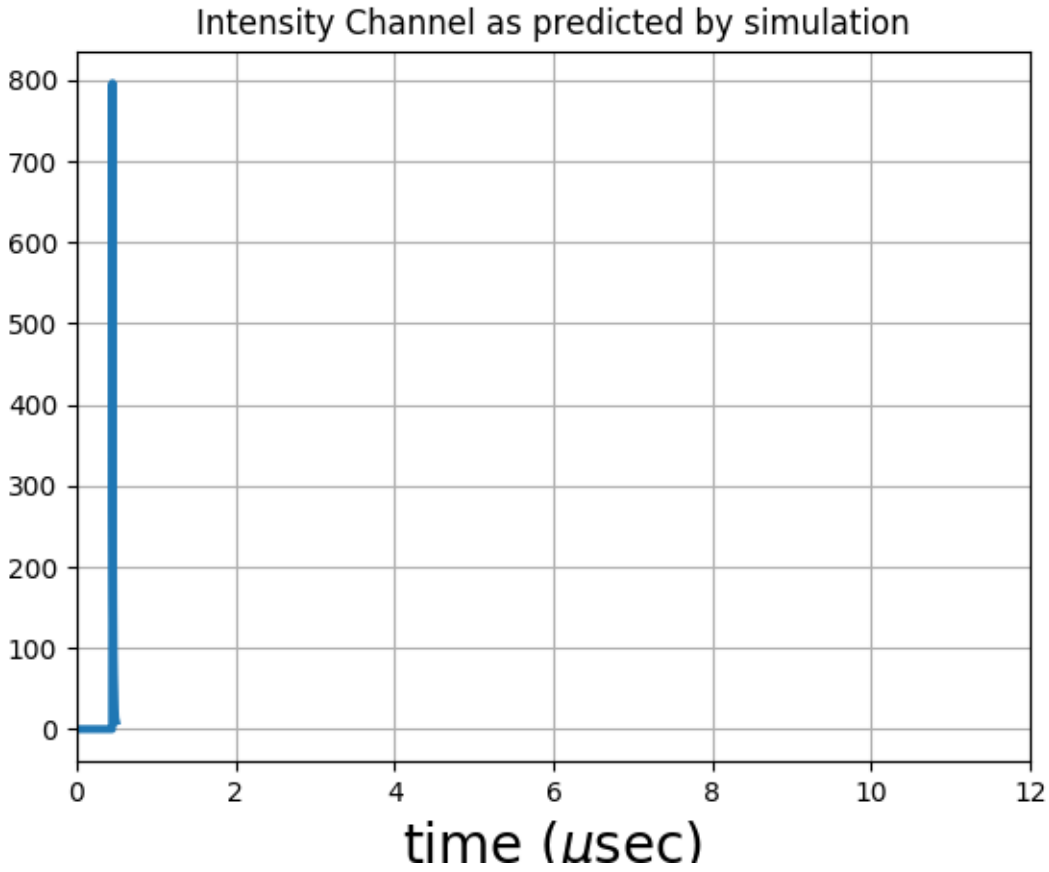
As the python code is very slow and it takes nearly 4 to 5s per loop. Running with 10002 time steps will take nearly 10hrs and hence I ran it with 100 time steps and 100000 rays.

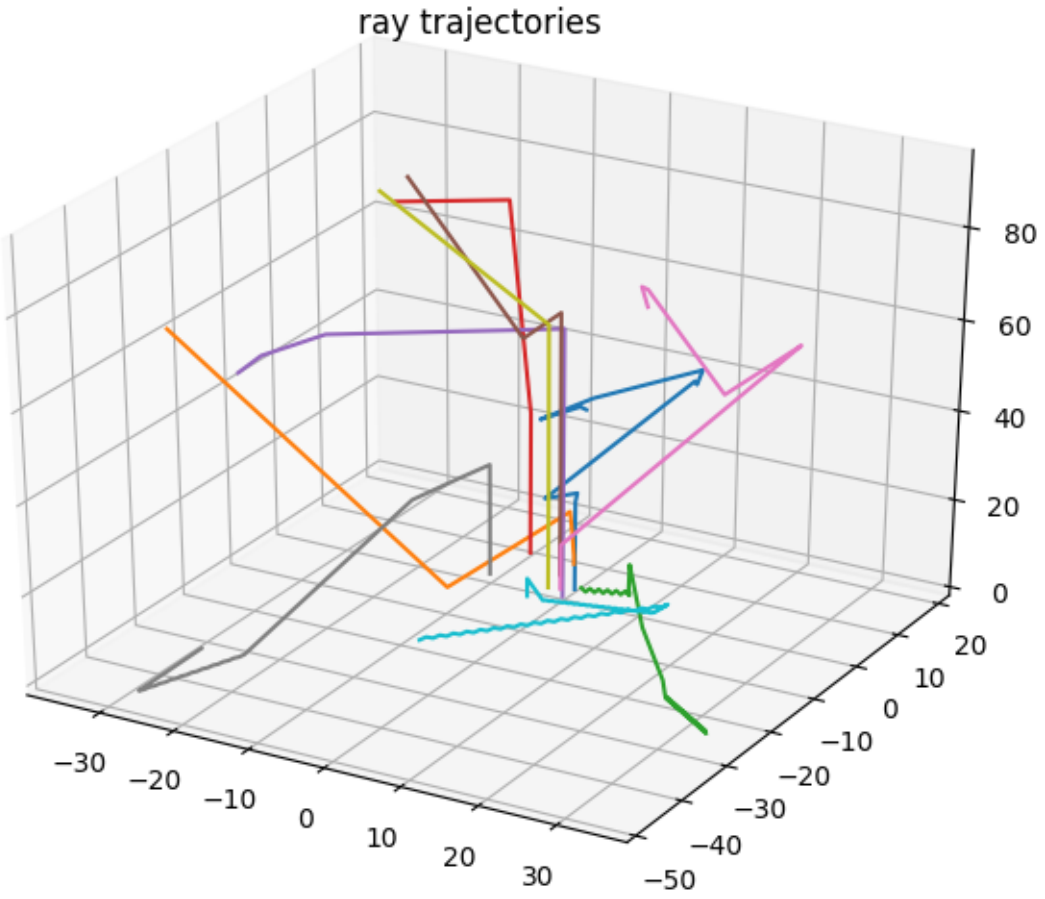
7.5.2 Output:

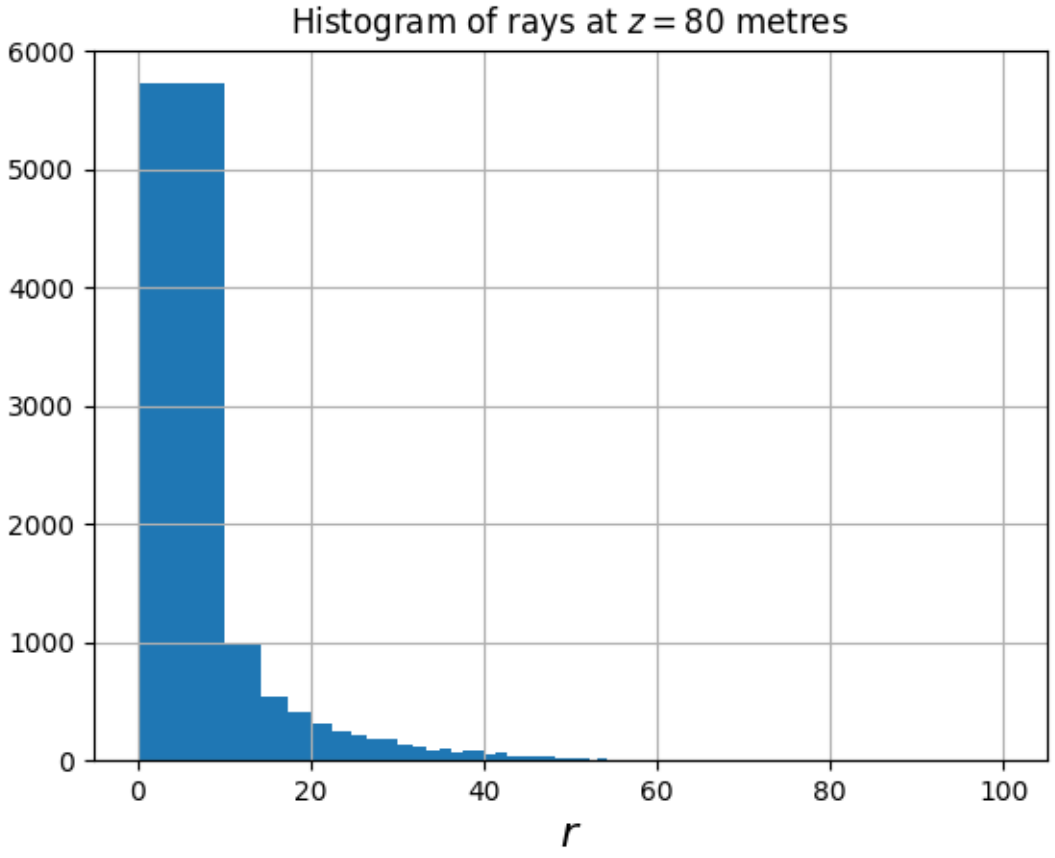
The output of the final code is:
6022 rays (out of 100000) reached the submarine
Average time to reach submarine=91.04 time steps
Stdeviation of time to reach submarine=1.69 time steps

7.6 Plots from Openacc






ray trajectories
80
60
40
20
0
20
10
0
−10
−20
−30
−40
−50
−30
−20
−10
0
10
20
30


Histogram of rays at $z = 80$ metres
6000
5000
4000
3000
2000
1000
0
0
20
40
60
80
100
r

7.6.1 Conditions:

The openacc code takes nearly 0.015s per loop. To match the output with python, I ran it for 100 time steps and 100000 rays. Also I have assumed constant extrapolation and therefore there will be some difference in the plots and output.

7.6.2 Output:

The output of the final code is:
3541 rays (out of 100000) reached the submarine
Average time to reach submarine=91.17 time steps
Stdeviation of time to reach submarine=1.80 time steps

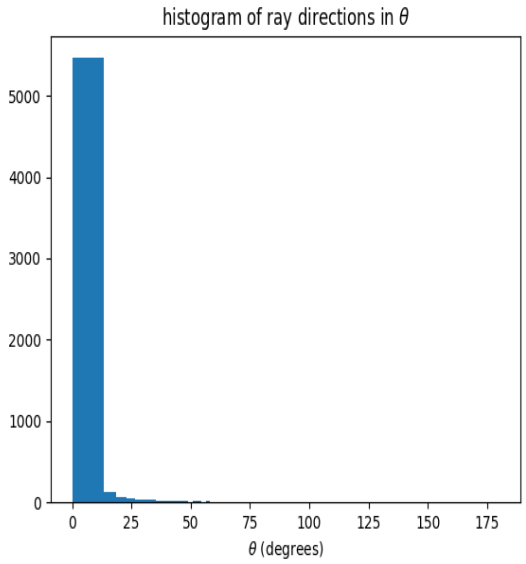
But if I keep index constant, the results match with those of python and then the output is:
6032 rays (out of 100000) reached the submarine
Average time to reach submarine=91.03 time steps
Stdeviation of time to reach submarine=1.71 time steps

Also if I run it for 10002 time steps then the output is:
7213 rays (out of 100000) reached the submarine
Average time to reach submarine=1119.79 time steps
Stdeviation of time to reach submarine=2042.26 time steps

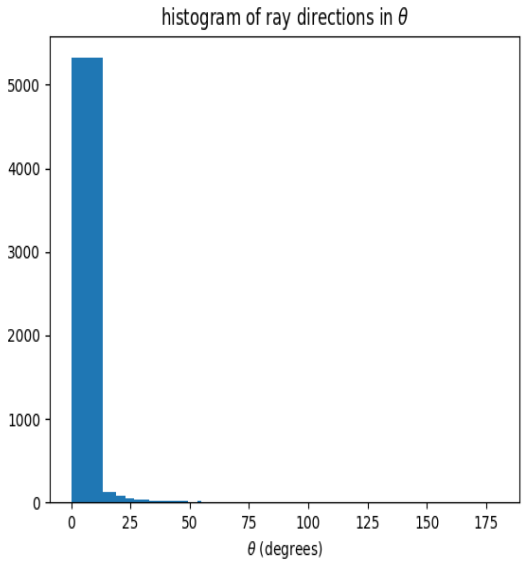
8 Comparison:

There are three subsection in this section. In the first two sections i.e. with 100 and 1000 iterations, I have included the plots for all the cases but for 10002 iterations, I have excluded the Python case as it takes very long time to simulate. Also in the first two cases where the python plots have been included, I have kept the index constant so as to match with Python.

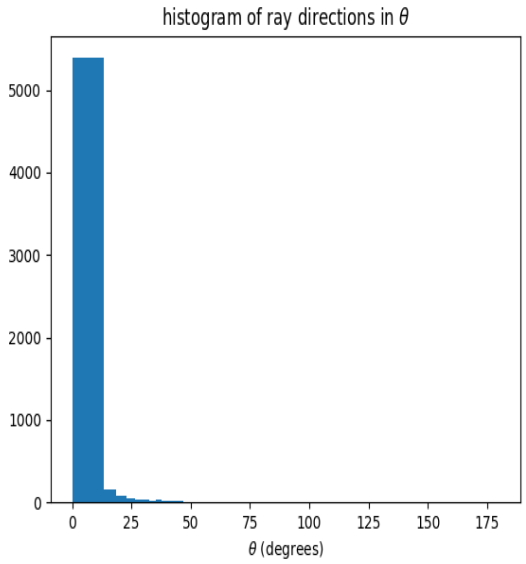
8.1 Comparison with 100 iterations:



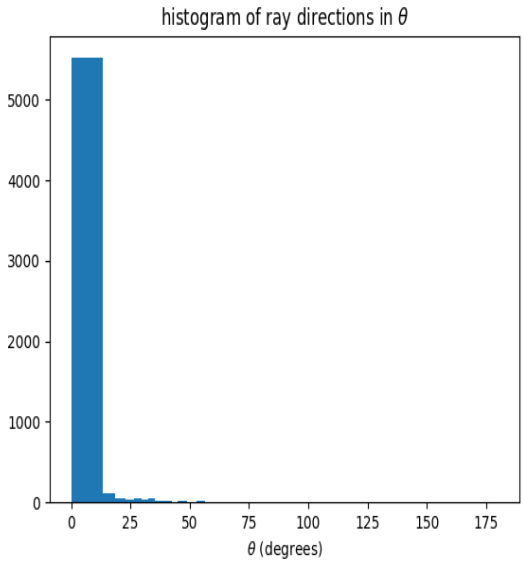
(a) Python



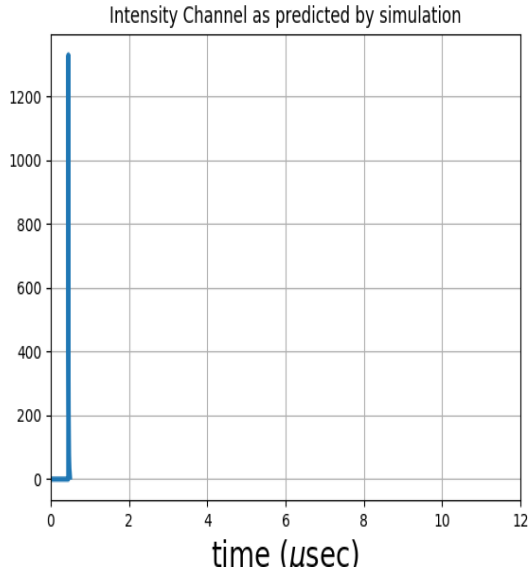
(b) C



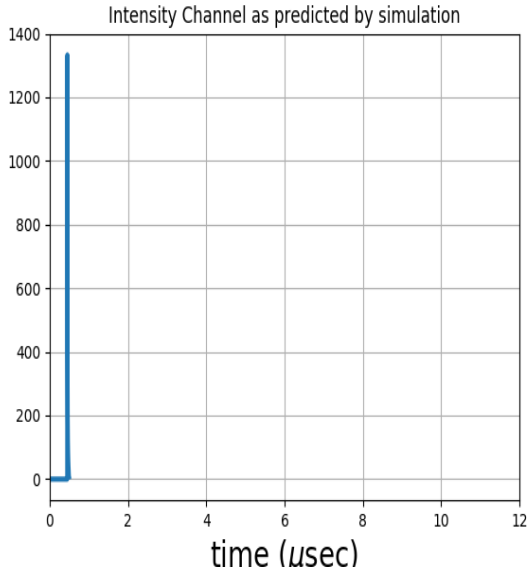
(c) OpenMp



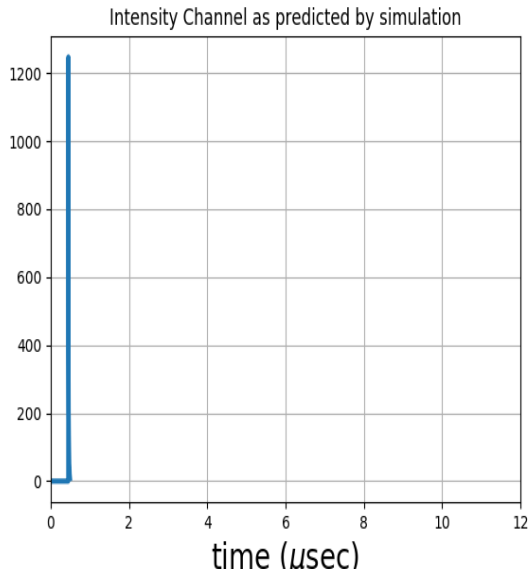
(d) Openacc



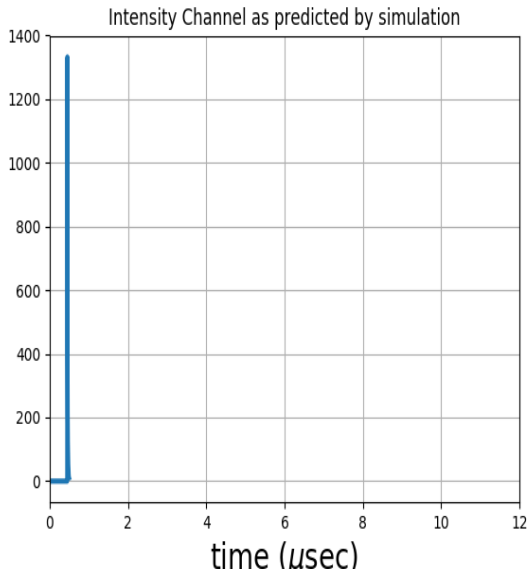
(a) Python



(b) C

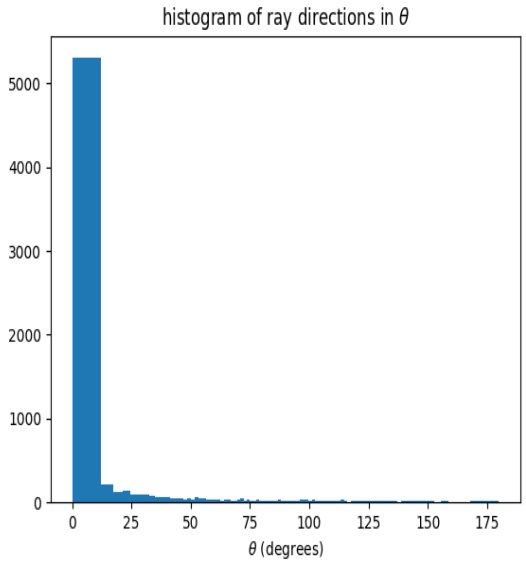


(c) OpenMp

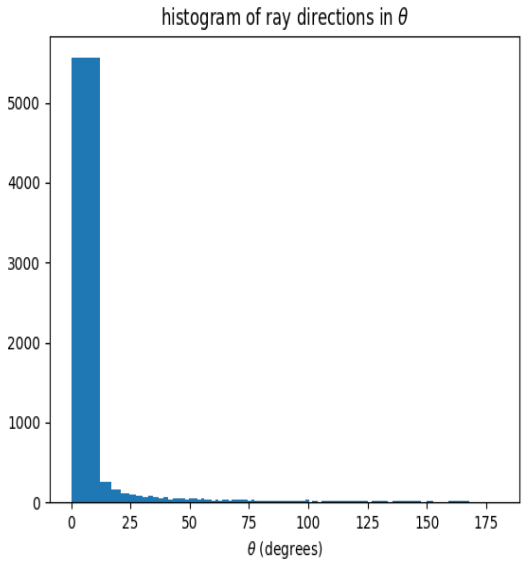


(d) Openacc

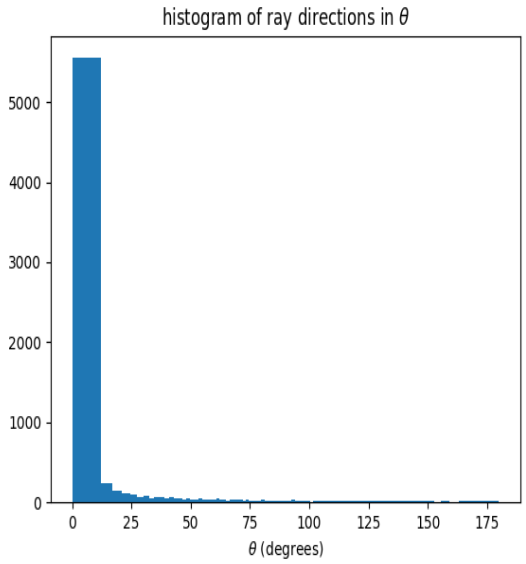
8.2 Comparison with 1000 iterations:



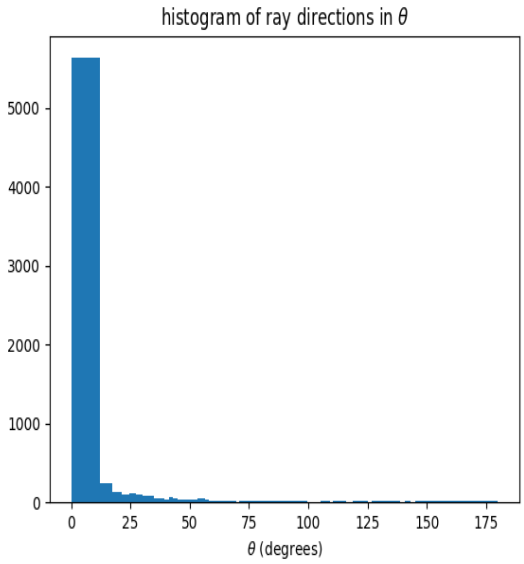
(a) Python



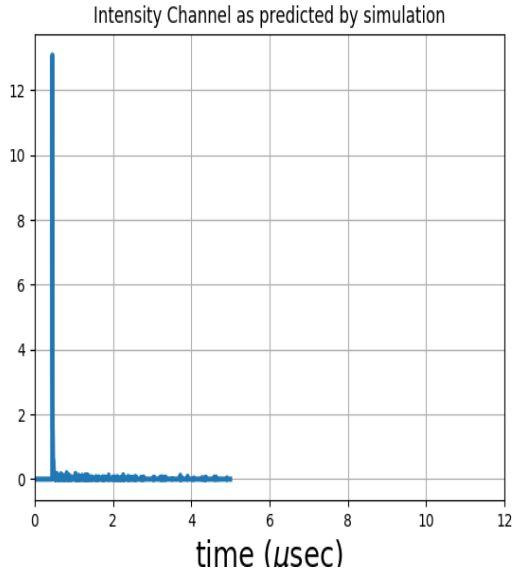
(b) C



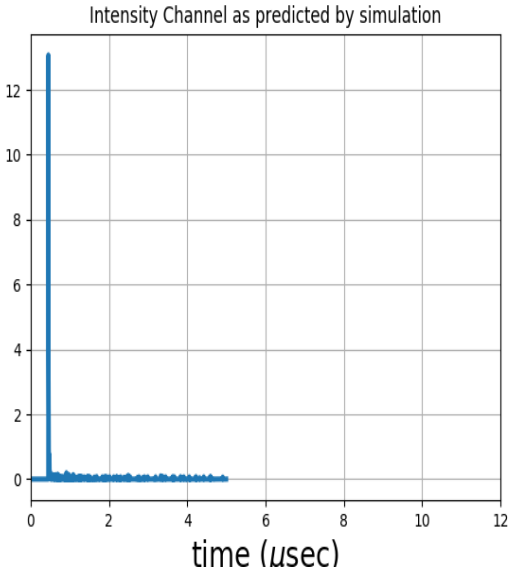
(c) OpenMp



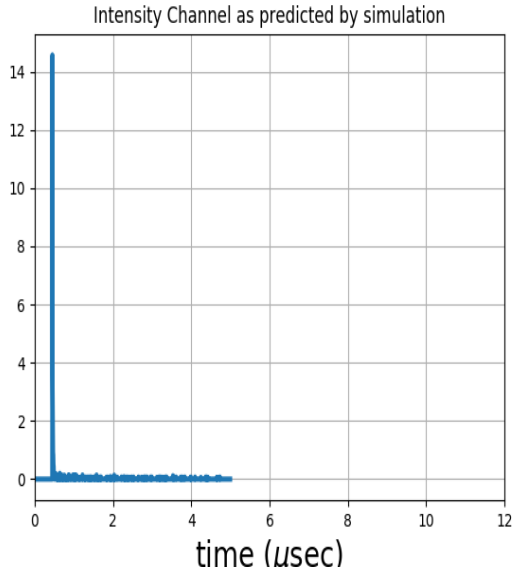
(d) Openacc



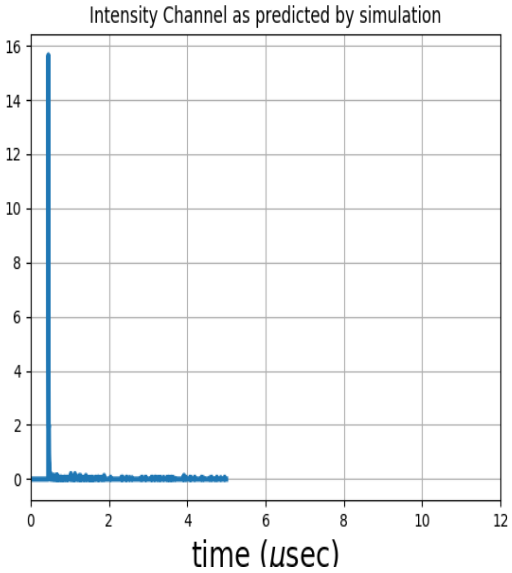
(a) Python



(b) C

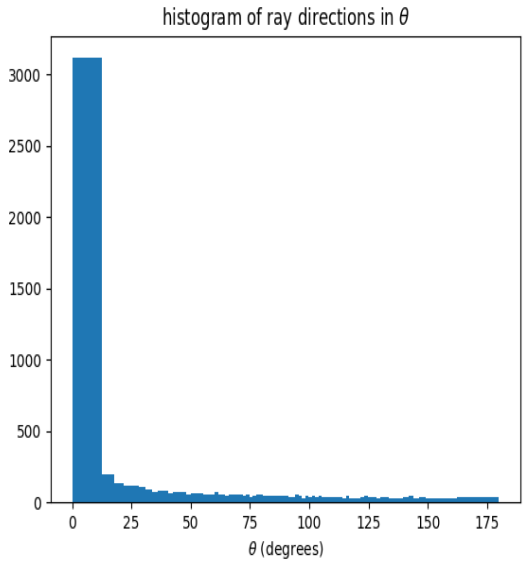


(c) OpenMp

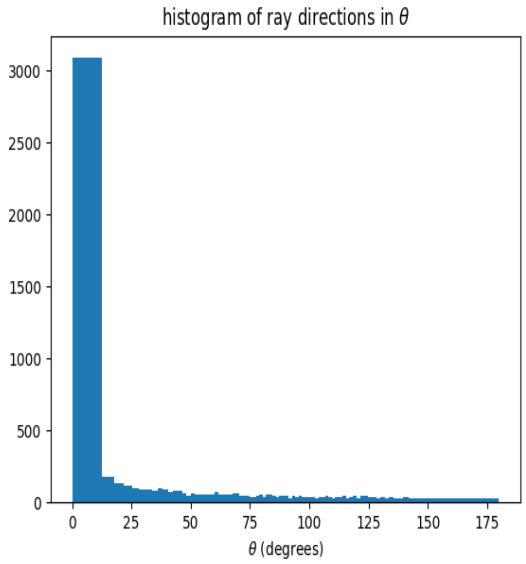


(d) Openacc

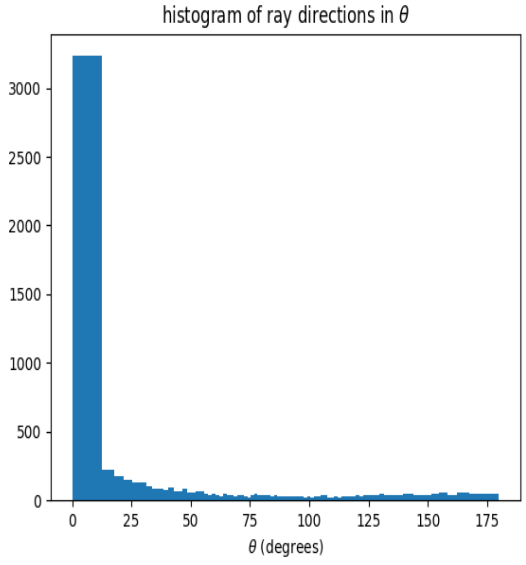
8.3 Comparison with 10002 iterations:



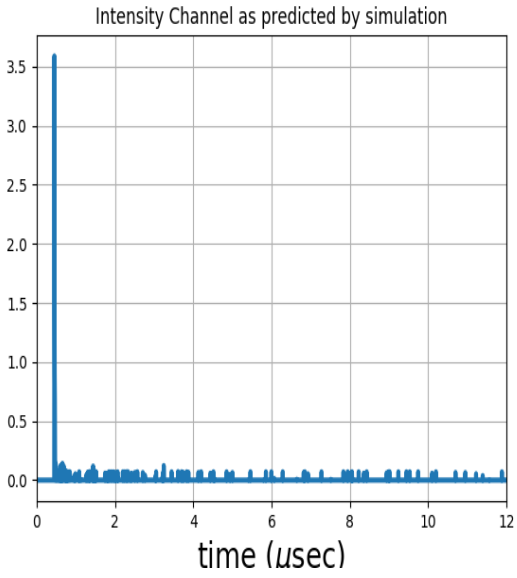
(a) C



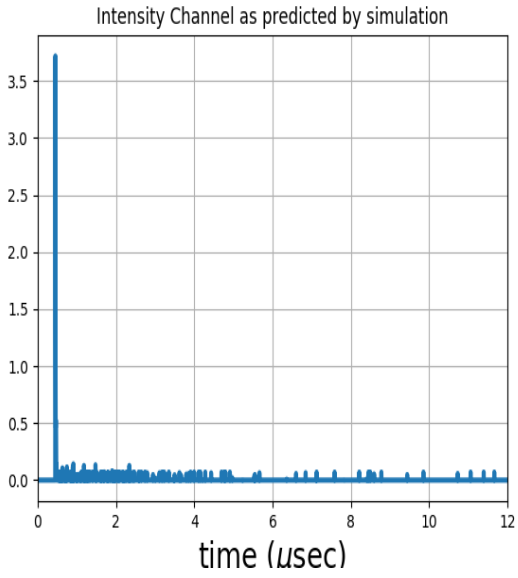
(b) OpenMp



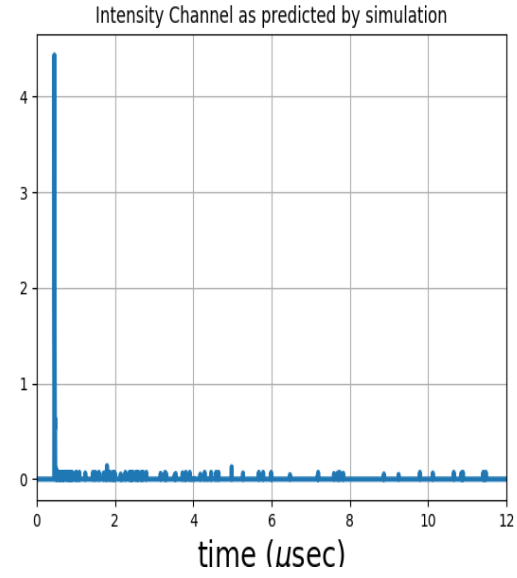
(c) Openacc



(a) C



(b) OpenMp



(c) Openacc

9 References:

[1] Dr. Yogish Sabharwa. (2017, June 8). *Introduction to Parallel Programming in OpenMp* [Online]. Available: http://nptel.ac.in/courses/106102163/

[2] Dr. Subodh Kumar.(2013, Nov 14). *Parallel Computing* [Online]. Available: http://nptel.ac.in/courses/106102114/

[3] Cscsch. (2013, Dec 18). *An introduction to OpenACC* [Online]. Available: https://www.youtube.com/watch?v=KQ0SOx46Xf0

[4] Jeff Larkin. (2015, Oct 1). *Introduction to OpenACC on x86 CPU and GPU* [Online]. Available: https://www.youtube.com/watch?v=KgMJzmqenuclist=PL5B692fm6–u-tdH8ct52nXWqmmp4B3C-index=1

[5] Wikipedia. *Parallel computing* [Online]. Available: https://en.wikipedia.org/wiki/Parallel_computing

Appendix

A Ray tracing C code

This is the main part of my project where I have converted the python code into c and then ported it to GPU. I will explain each block of C code.

Listing 12: Ray tracing

```
#include<time.h>
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#include<openacc.h>

#define PI 3.14159265358979323846

#pragma acc routine seq
    unsigned long rnd(long prev){
        unsigned long a = 1103515245,m = 2147483648,c = 12345;
        unsigned long nt = a*prev+c;
        nt = nt%m;
        return nt;
    }

double randn (double mu, double sigma){
    double U1, U2, W, mult;
    static double X1, X2;
    static int call = 0;

    if (call == 1) {
        call = !call;
        return (mu + sigma * (double) X2);
    }

    do{
        U1 = -1 + ((double) rand () / RAND_MAX) * 2;
        U2 = -1 + ((double) rand () / RAND_MAX) * 2;
        W = (double) pow(U1, 2) + (double) pow(U2, 2);
    }while (W >= 1 || W == 0);
```

```
33
34      mult = sqrt ((-2 * log (W)) / W);
35      X1 = U1 * mult;
36      X2 = U2 * mult;
37      call = !call;
38
39      return (mu + sigma * (double) X1);
40  }
```

In the above section there are some libraries which are imported and Gaussian random function which takes two parameters that are mu(mean) and sigma(standard deviation). As there is no inbuilt function in c for normalized random function, I had to define it here. Also as we cannot use inbuilt rand() function in openacc, I had to define a pseudo random generator function which can be paralleled.

Listing 13: Spline & Splint

```
1  //#pragma acc routine seq
2  void spline(double *x,double *y,int n,double yp1,double ypn,double ↩
       *y2/*,double *u*/){
3      // It is assumed that x,y,y2 and u have been allocated in the ↩
           calling program. u is a work array of the same size as x.
4      int i,k;
5      double p,qn,sig,un,u[n+1];
6
7      x--;y--;y2--; // NR adjustments
8      if (yp1 > 0.99e30) y2[1]=u[1]=0.0;
9      else {
10         y2[1] = -0.5;
11         u[1]=(3.0/(x[2]-x[1]))*((y[2]-y[1])/(x[2]-x[1])-yp1);
12     }
13     for (i=2;i<=n-1;i++) {
14         sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
15         p=sig*y2[i-1]+2.0;
16         y2[i]=(sig-1.0)/p;
17         u[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-x[i↩
               -1]);
18         u[i]=(6.0*u[i]/(x[i+1]-x[i-1])-sig*u[i-1])/p;
19     }
20     if (ypn > 0.99e30) qn=un=0.0;
21     else {
```

```
22          qn=0.5;
23          un=(3.0/(x[n]-x[n-1]))*(ypn-(y[n]-y[n-1])/(x[n]-x[n-1]));
24      }
25      y2[n]=(un-qn*u[n-1])/(qn*y2[n-1]+1.0);
26      for (k=n-1;k>=1;k--){
27          y2[k]=y2[k]*y2[k+1]+u[k];
28      }
29  }
30
31
32  //#pragma acc routine seq
33  double splint(double *xa,double *ya,double *y2a,int n,double x){
34      // void nrerror();
35      int klo,khi,k;
36      double h,b,a,y;
37
38      xa--;ya--;y2a--; // NR adjustments
39      klo=1;
40      khi=n;
41      while (khi-klo > 1) {
42          k=(khi+klo) >> 1;
43          if (xa[k] > x) khi=k;
44          else klo=k;
45      }
46      h=xa[khi]-xa[klo];
47      if (h == 0.0){
48          printf("Bad xa input to routine splint");
49          exit(1);
50      }
51      a=(xa[khi]-x)/h;
52      b=(x-xa[klo])/h;
53      y=0.0+a*ya[klo]+b*ya[khi]+( (a*a*a-a)*y2a[klo]+(b*b*b-b)*y2a[↩
            khi] )*(h*h)/6.0;
54      return y;
55  }
```

Spline and Splint function are taken from Numerical Recipes. Spline function takes coordinates, number of coordinates and first order derivative at the first and last coordinate as inputs. In return the function calculates the second order derivatives at all the coordinates. Whereas Splint takes the coordinates, second order derivatives and one

x-coordinate as input and it returns the corresponding y-value.

This spline produces a curve which passes through all the coordinates and is good for interpolation of any curve. But for extrapolation, the error becomes large and hence I have considered two cases i.e. let the function extrapolate in one case while in the other, taking constant value for extrapolation.

Listing 14: Extra function for Spline & Splint in open acc

```
1
2  /*These two functions are only meant for openacc*/
3
4
5  #pragma acc routine seq
6      void spline1(double *x,double y0,double y1,double y2,double y3,↩
          double y4,double y5,double y6,double y7,int n,double yp1,↩
          double ypn,double *y2_0,double *y2_1,double *y2_2,double *↩
          y2_3,double *y2_4,double *y2_5,double *y2_6,double *y2_7){
7          // It is assumed that x,y,y2 and u have been allocated in ↩
              the calling program. u is a work array of the same size ↩
              as x.
8          int i,k;
9          double p,qn,sig,un;
10         //double *u = malloc(sizeof(double)*110);
11         double u0,u1,u2,u3,u4,u5,u6,u7;
12         //x--;y--;y2--; // NR adjustments
13         if (yp1 > 0.99e30) *y2_0=u0=0.0;
14         else {
15             *y2_0 = -0.5;
16             u0=(3.0/(x[1]-x[0]))*((y1-y0)/(x[1]-x[0])-yp1);
17         }
18         i=1;
19         sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
20         p=sig*(*y2_0)+2.0;
21         *y2_1=(sig-1.0)/p;
22         u1=(y2-y1)/(x[i+1]-x[i]) - (y1-y0)/(x[i]-x[i-1]);
23         u1=(6.0*(u1)/(x[i+1]-x[i-1])-sig*(u0))/p;
24
25         i=2;
26         sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
27         p=sig*(*y2_1)+2.0;
28         *y2_2=(sig-1.0)/p;
```

```
29          u2=(y3-y2)/(x[i+1]-x[i]) - (y2-y1)/(x[i]-x[i-1]);
30          u2=(6.0*(u2)/(x[i+1]-x[i-1])-sig*(u1))/p;
31
32          i=3;
33          sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
34          p=sig*(*y2_2)+2.0;
35          *y2_3=(sig-1.0)/p;
36          u3=(y4-y3)/(x[i+1]-x[i]) - (y3-y2)/(x[i]-x[i-1]);
37          u3=(6.0*(u3)/(x[i+1]-x[i-1])-sig*(u2))/p;
38
39          i=4;
40          sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
41          p=sig*(*y2_3)+2.0;
42          *y2_4=(sig-1.0)/p;
43          u4=(y5-y4)/(x[i+1]-x[i]) - (y4-y3)/(x[i]-x[i-1]);
44          u4=(6.0*(u4)/(x[i+1]-x[i-1])-sig*(u3))/p;
45
46          i=5;
47          sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
48          p=sig*(*y2_4)+2.0;
49          *y2_5=(sig-1.0)/p;
50          u5=(y6-y5)/(x[i+1]-x[i]) - (y5-y4)/(x[i]-x[i-1]);
51          u5=(6.0*(u5)/(x[i+1]-x[i-1])-sig*(u4))/p;
52
53          i=6;
54          sig=(x[i]-x[i-1])/(x[i+1]-x[i-1]);
55          p=sig*(*y2_5)+2.0;
56          *y2_6=(sig-1.0)/p;
57          u6=(y7-y6)/(x[i+1]-x[i]) - (y6-y5)/(x[i]-x[i-1]);
58          u6=(6.0*(u6)/(x[i+1]-x[i-1])-sig*(u5))/p;
59
60          if (ypn > 0.99e30) qn=un=0.0;
61          else {
62              qn=0.5;
63              un=(3.0/(x[n-1]-x[n-2]))*(ypn-(y7-y6)/(x[n-1]-x[n-2]));
64          }
65
66          *y2_7=(un-qn*(u6))/(qn*(*y2_6)+1.0);
67
68          *y2_6=*y2_6*(*y2_7)+u6;//k=6
69          *y2_5=*y2_5*(*y2_6)+u5;//k=5
```

```
70          *y2_4=*y2_4*(*y2_5)+u4;//k=4
71          *y2_3=*y2_3*(*y2_4)+u3;//k=3
72          *y2_2=*y2_2*(*y2_3)+u2;//k=2
73          *y2_1=*y2_1*(*y2_2)+u1;//k=1
74          *y2_0=*y2_0*(*y2_1)+u0;//k=0
75
76      }
77
78
79  #pragma acc routine seq
80      double splint1(double *xa,double ya0,double ya1,double ya2,↩
           double ya3,double ya4,double ya5,double ya6,double ya7,↩
           double y2a_0,double y2a_1,double y2a_2,double y2a_3,double ↩
           y2a_4,double y2a_5,double y2a_6,double y2a_7,int n,double x)↩
           {
81          // void nrerror();
82          int klo,khi,k;
83          double h,b,a,y;
84
85          if(x<xa[0]) return ya0;
86          else if (x>xa[n-1]) return ya7;
87
88          //xa--;ya--;y2a--; // NR adjustments
89          klo=0;
90          khi=n-1;
91          while (khi-klo > 1) {
92              k=(khi+klo) >> 1;
93              if (xa[k] > x) khi=k;
94              else klo=k;
95          }
96          h=xa[khi]-xa[klo];
97          if (h == 0.0){
98              return 0;
99              //printf("Bad xa input to routine splint");
100             //exit(1);
101         }
102         a=(xa[khi]-x)/h;
103         b=(x-xa[klo])/h;
104         double t1,t2,t3,t4;
105         switch(klo){
106             case 0:
```

```
                t1=ya0;
                t3=y2a_0;
                break;
            case 1:
                t1=ya1;
                t3=y2a_1;
                break;
            case 2:
                t1=ya2;
                t3=y2a_2;
                break;
            case 3:
                t1=ya3;
                t3=y2a_3;
                break;
            case 4:
                t1=ya4;
                t3=y2a_4;
                break;
            case 5:
                t1=ya5;
                t3=y2a_5;
                break;
            case 6:
                t1=ya6;
                t3=y2a_6;
                break;
            case 7:
                t1=ya7;
                t3=y2a_7;
                break;
        }
        switch(khi){
            case 0:
                t2=ya0;
                t4=y2a_0;
                break;
            case 1:
                t2=ya1;
                t4=y2a_1;
                break;
```

```
148             case 2:
149                 t2=ya2;
150                 t4=y2a_2;
151                 break;
152             case 3:
153                 t2=ya3;
154                 t4=y2a_3;
155                 break;
156             case 4:
157                 t2=ya4;
158                 t4=y2a_4;
159                 break;
160             case 5:
161                 t2=ya5;
162                 t4=y2a_5;
163                 break;
164             case 6:
165                 t2=ya6;
166                 t4=y2a_6;
167                 break;
168             case 7:
169                 t2=ya7;
170                 t4=y2a_7;
171                 break;
172         }
173         //y=0.0+a*ya[klo]+b*ya[khi]+( (a*a*a-a)*y2a[klo]+(b*b*b-b)*↩
               y2a[khi] )*(h*h)/6.0;
174         y=0.0+a*t1+b*t2+( (a*a*a-a)*t3+(b*b*b-b)*t4 )*(h*h)/6.0;
175         return y;
176     }
```

I had to write these alternate versions of spline and splint specifically to find values of theta0. I was not able to privatize an array which either gave me wrong results or segmentation faults. Therefore I defined these functions so as to remove any privatization of array per thread instead now I have to privatize a bunch of variables.

Listing 15: Read File

```
1 int readfile(char* name,int n,double mat[][n],int max_r){
2     char buffer[1024];
3     char *record,*line;
```

```
4       int i=0,j=0;
5       FILE *fstream = fopen(name,"r");
6       if(fstream == NULL){
7           printf("\n file opening failed ");
8           return 0;
9       }
10      /*else{
11          printf("\n file opened ");
12      } */
13      while((line=fgets(buffer,sizeof(buffer),fstream))!=NULL)
14      {
15          record = strtok(line,",");
16          if(record[0] == '#') continue;
17          while(record != NULL)
18          {
19              //here you can put the record into the array as per ↩
                    your requirement.
20              mat[i][j++] = atof(record);
21              record = strtok(NULL,",");
22              if(j==n) break;
23          }
24          ++i;
25          if(max_r!=0 && i>=max_r) break;
26          j=0;
27      }
28      return i;
29  }
```

The function is used to read file. It takes file name, number of columns and number of rows to read and an address to a 2-D array to store the values. The function is designed in such a way that there is no need to read complete file and then take a smaller matrix out of it. By giving correct dimensions, the function reads only smaller matrix and skips other part with certain constraint like the left-top dimension of the matrix cannot be changed. Also if a line starts with "#", it considers as a comment and skips it as in python.

Listing 16: Integral

```
1   double intgrl(double *theta,double *sangle,int ni,int nj,double ↩
        Sca_T[nj][ni],int n,double sc[8][98],int j,double start, double ↩
        end){
2       double r2deg = 180.0/PI;
```

```
    int N = 1000; //@@Increase it for accuracy.
    double i, inc = (end-start)/N, sum = 0;
    double a = splint(sangle,(double *)Sca_T +ni*j+1,(double *)sc+(↩
       j-1)*(ni-1),n,start*r2deg)*2*PI*sin(start);
    double b;
    start+=inc;

    for (i = start; i < end; i += inc) {
        b = splint(sangle,(double *)Sca_T+ni*j+1,(double *)sc+(j-1)↩
           *(ni-1),n,i*r2deg)*2*PI*sin(i);
        sum += 0.5 * inc * (a+b);
        a = b;
    }
    return sum;
}
```

To replicate the quad function in Python, I have written this integral function. The integration principle is based on finding the area of smaller trapezium and summing it all. The points of trapezium are x, x+dx and their respective values at that points. The accuracy of the function can be increased by decreasing dx and that can be done by increasing N, which is inversely proportional to dx.

Listing 17: Main

```
int main(int argc, char const *argv[])
{
    /* code */
    clock_t start, end,p_start,p_end,p_st;
    double cpu_time_used;


    int nt = 250;  /*# number of time steps to simulate. By this ↩
       time, active
    # rays have decayed to exp(-attn*nt) of its initial value.
    # Choose nt based on this.
    # for reasons I don't remember, nt is not a round number*/
    int z0 = 100; /* # centre of submarine*/
    int s = 10;  /*  # size of submarine (cube)*/
    int ntraj = 10;/* # number of trajectories to track for ↩
       plotting*/

```

```
16      //srand(time(0)); # randomize the random number generator
17      /*# Entering beam is assumed to have an intensity that is
18      # spatially normally distributed in r with stdeviation of r0.*/
19      int r0 = 5; /*   # size of the input beam region*/
20      int wavelength=513; /*# nanometers*/
21      /*# file containing attenuation information*/
22      char fattenuation[] = "data/↩
           absorption_coefficient_april2017_T3D3.csv";
23      double attnConst = 0.001;
24      double attn; //Introduced by me. Removing the function
25
26      //@@Only else block
27      int i,j,n_att = 12,n;
28      double yp1,ypn;
29      double Att[351][13]; //Values from the file ↩
           absorption_coefficient
30      double y2_att[n_att]; //Contains 2nd order derivative
31      double y_att[n_att],x_att[n_att]; //Y and X coordinates ↩
           respectively
32      int row = readfile(fattenuation,13,Att,0);
33      for(i=0;i<row;i++){
34          if(Att[i][0] == wavelength){
35              for(j=0;j<n_att;j++){
36                  x_att[j] = Att[0][j+1];
37                  y_att[j] = Att[i][j+1];
38              }
39              yp1 = (Att[i][2]-Att[i][1])/(Att[0][2]-Att[0][1]);
40              ypn = (Att[i][n_att]-Att[i][n_att-1])/(Att[0][n_att]-↩
                   Att[0][n_att-1]);;
41              //spline((double *)Att+1,(double *)Att+13*i+1,n_att,yp1↩
                   ,ypn,y2_att);
42              spline(x_att,y_att,n_att,yp1,ypn,y2_att);
43              break;
44          }
45      }
```

Now the main function starts where initially all the variables have been declared. Then it reads the absorption coefficient file. The file contains the attenuation values for different wavelength and based on the wavelength provided, it calls the spline function w.r.t the values in the given wavelength row.

Listing 18: Reading Scattering Data

```
/*# fscattering contains the file name for scattering. If blank,
    # model is used.
    #fscatname="VSF-april2017-T3D3"*/
    char fscatname[] = "VSF-sept2017-T2D3";
    //# fscatname = "VSF-sept2017-T2D1"
    char fscattering[] = "data/VSF-sept2017-T2D3.csv";

    int ni,nj;
    int Nc = 99;
    ni = Nc;
    nj = 9;

    double Sca[ni][nj];
    double Sca_T[nj][ni];//Transpose of Sca matrix
    double sdepth[nj-1],sangle[ni-1],stheta[ni-1],dtheta[ni-1];
    double r2deg = 180.0/PI,angle[ni],theta[ni];
    double sc[nj-1][ni-1];//Contains 2nd order derivative

    if (strcmp(fscattering,"") != 0){

        char fsflag[] = "scatdata";
        //Scb=loadtxt(fscattering,delimiter=",")

        row = readfile(fscattering,nj,Sca,Nc);

        angle[0] = 0.0;theta[0] = 0.0;
        dtheta[0] = 0;dtheta[ni-2] = 0;
        for(i=0;i<nj-1;i++){
            sdepth[i] = Sca[0][i+1];
        }
        for(i=0;i<ni-1;i++){
            sangle[i] = Sca[i+1][0];
            stheta[i] = Sca[i+1][0]*PI/180.0;
            angle[i+1] = Sca[i+1][0];
            theta[i+1] = Sca[i+1][0]*PI/180.0;
            if(i>=2){
                dtheta[i-1] = (stheta[i]-stheta[i-2])*0.5;
            }
        }
```

In the second section of the main function, it reads the scattering file and store the values in the Sca matrix. In the python code, the whole file was read and then it was cropped to (ni,nj). To save time, I didn't read it at all. After reading it extracts the proper rows and columns from the Sca matrix and stores in different arrays like angle, sangle, stheta, theta,etc.

Listing 19: Scattering spline

```
1
2         for(i=0;i<ni;i++){
3             for(j=0;j<nj;j++){
4                 Sca_T[j][i] = Sca[i][j];
5             }
6         }
7
8         n = 98;
9         for(j=1;j<nj;j++){
10            yp1 = (Sca[2][j]-Sca[1][j])/(sangle[1]-sangle[0]);
11            ypn = (Sca[Nc-1][j]-Sca[Nc-2][j])/(sangle[Nc-2]-sangle[↩
                  Nc-3]);
12            spline(sangle,(double *)Sca_T+(Nc*j)+1,n,yp1,ypn,(↩
                  double *)sc+(j-1)*(Nc-1));
13        }
14    }
```

I have transposed Sca matrix and stored in Sca_T matrix to use the matrix as pointer in spline calls. It is because spline function requires continuous memory address and a column in Sca in not memory-contiguous. It saves a lot of time as by doing this there will be cache hits. The second order derivatives are stored in sc variable whereas in python, sc contains spline functions. The problem faced here was to avoid an extra loop to pass the y coordinates in spline function which was avoided by sending correct pointer address of the transposed matrix.

Listing 20: CDF

```
1 double P[nj][ni];
2     for(int temp1 = 0;temp1<nj;temp1++){
3         for(int temp2 = 0;temp2<ni;temp2++){
4             P[temp1][temp2] = 0.0;
5         }
6     }
```

```
7       //# Pnorm=zeros(P.shape)
8       printf("Starting quad block\n");
9
10      for(j = 0;j<nj-1;j++){
11          for(i=0;i<ni-1;i++){
12              P[j][i+1] = intgrl(theta,sangle,ni,nj,Sca_T,n,sc,j+1,↩
                    stheta[0],stheta[i]);
13          }
14          for(i=1;i<ni;i++){
15              P[j][i] = P[j][i] + 1 - P[j][ni-1];
16          }
17      }
18
19      FILE *fp1;
20      fp1 = fopen("P.csv", "w");
21      if(fp1 == NULL){
22          printf("Error!");
23          exit(1);
24      }
25
26      for(int i2=0;i2<nj;i2++){
27          for(int j2 = 0;j2<ni;j2++){
28              fprintf(fp1, "%lf",P[i2][j2]);
29              if(j2!=(ni-1)) fprintf(fp1, ",");
30          }
31          fprintf(fp1, "\n");
32      }
33      fclose(fp1);
```

The variable P is the CDF curve. To fill in each cell of the 2-D array, integral function is called with the function and start and end value. The intgrl function then calls the splint function to get the value at that point and then it integrates the curve and returns a double value i.e. the output. Now after that in the second for loop, it opens a file named "P.csv" and writes the array in that csv file. It will be useful in plotting the CDF in python where a python file reads from the csv file and do the necessary plotting. Also the values in P array are not efficiently filled. As we can see that P[j][i+1] requires integration from stheta[0] to stheta[i] and P[j][i+2] requires integration from stheta[0] to stheta[i+1]. But the code is written in such a way that to find P[j][i+2] it integrates from the beginning. Instead it can be written as:

$$\text{P[j][i+2]} = \text{P[j][i+1]} + \int_{stheta[i]}^{stheta[i+1]} sc[j](theta * r2deg) * 2 * pi * sin(theta) dtheta$$

Listing 21: Inverse CDF

```
    n = 98;

    double Pinv[nj-1][n];
    double tt,temp[Nc-1],temp2[Nc-1];
    for(j = 0;j<nj-1;j++){
        yp1 = (angle[2]-angle[1])/(P[j][2]-P[j][1]);
        ypn = (angle[Nc-1]-angle[Nc-2])/(P[j][Nc-1]-P[j][Nc-2]);

        spline((double *)P+Nc*j+1,(double *)angle+1,n,yp1,ypn,(↩
           double *)Pinv+j*n);

    }
```

In Python, Pinv variable was a function pointer to inverse CDF. But in C it contains second order derivative values so as to reduce the number of spline calls. yp1 and ypn are the first order derivative at the initial and last points. I have assigned it as the slope between the first and second point. Having the second order derivatives, it needs a splint call to get the output. If we observe the spline function is called from 2nd column elements. So, when using a splint function, we should make sure that the input must lie within its range and it shouldn't extrapolate. So a if condition is required which ensures the above statement. Also the other difference is that in Python, it was marked by a if-else condition which I have eliminated in here. I have added the if-else condition whenever the function was called.

Listing 22: Declaration and Allocation of Variables

```

    //# Allocate arrays for storing the results
    int N = 100000;// # number of rays in all

    double *pos[3];// # tracks position of rays
    for(int i=0;i<3;i++){
        pos[i] = (double*)malloc(N*sizeof(double));
    }

```

```
10      double *direction[3];// # remembers dirction of ray
11      for(int i=0;i<3;i++){
12          direction[i] = (double*)malloc(N*sizeof(double));
13      }
14
15      double *intensity = (double*)malloc(N*sizeof(double));//  # ↩
           intensity of ray initially one.
16      int *tsrc = (int*)malloc(N*sizeof(int));;//    # time at which ↩
           ray was born.
17      //intensity = (double*)malloc(N*sizeof(double));
18
19
20      double *flx[3];//   # direction from which the ray hit the ↩
           submarine.
21      for(int i=0;i<3;i++){
22          flx[i] = (double*)malloc(N*sizeof(double));
23      }
24      int *status = (int*)malloc(N*sizeof(int));;//  # if zero active↩
           , if positive, reached sub at that time.
25
26      double channel[nt];// # will hold the channel
27      //double traj[3][ntraj][nt];//=zeros((3,ntraj,nt));// # holds ↩
           the trajectories of a selected number of rays
28      /*# We store the positions of rays when they first
29      # cross z=z1 and z=z2. The arrays are initialized
30      # to an impossible number.*/
31      double ***traj;
32      traj = (double***)malloc(sizeof(double**)*3);
33      for(int i=0;i<3;i++){
34          traj[i] = (double**)malloc(sizeof(double*)*ntraj);
35          for(int j=0;j<ntraj;j++){
36              traj[i][j] = (double*)malloc(sizeof(double)*nt);
37          }
38      }
39
40      double **beam1;/*# holds the (r,theta) positions of rays when ↩
           they first cross z1.*/
41      beam1 = (double**)malloc(N*sizeof(double*));
42      for(int i=0;i<N;i++){
43          beam1[i] = (double*)malloc(2*sizeof(double));
44      }
```

```
45      int z1 = 0;
46      double **beam2;/*# holds the (r,theta) positions of rays when ↩
           they first cross z2*/
47      beam2 = (double**)malloc(N*sizeof(double*));
48      for(int i=0;i<N;i++){
49          beam2[i] = (double*)malloc(2*sizeof(double));
50      }
51
52
53      int z2 = z0-2*s; //Both the variables are ints and are defined ↩
           earlier
54      int rmax2 = 10000;
55      int nbins2 = 100;
56
57      int a2 = (nt*100) + 1000;
58      double **theta0vals;/*#to hold 100 random theta0
59                           # values per time step .*/
60      theta0vals = (double**)malloc(sizeof(double*)*a2);
61      for(int i=0;i<a2;i++){
62          theta0vals[i] = (double*)malloc(2*sizeof(double));
63      }
64
65
66      double *phi0 = (double*)malloc(sizeof(double)*N);
67      double *theta0 = (double*)malloc(sizeof(double)*N);
```

Here all the required variables with the required data types are declared. Now all the variables are allocated dynamic memory i.e. using malloc because of the memory constraint otherwise. All these variables will be freed or deallocated at the end of the simulation.

Listing 23: Initializing Variables

```
1
2       //# Main simulation loop
3       int k = 0;//  # keeps track of where we can insert new rays
4
5       for(int temp1=0;temp1<3;temp1++){
6           for(int temp2=0;temp2<ntraj;temp2++){
7               for(int temp3=0;temp3<nt;temp3++){
8                   traj[temp1][temp2][temp3] = 0.0;
```

```
            }
        }
    }

    for(int temp1=0;temp1<N;temp1++){
        phi0[temp1] = 0;
        theta0[temp1] = 0;
        status[temp1] = 0;
        tsrc[temp1] = 0;
        intensity[temp1] = 1;

        for(int temp2=0;temp2<2;temp2++){
            beam1[temp1][temp2] = -1.0;
            beam2[temp1][temp2] = -1.0;
        }

        for(int temp2=0;temp2<3;temp2++){
            //pos[temp2][temp1] = 0;
            direction[temp2][temp1] = 0;
            //flx[temp2][temp1] = 0;
        }

    }

    double phi1,r;
    FILE *fpt,*fp;

    for(int i2=0;i2<N;i2++){
        phi1= (1.0*rand()/RAND_MAX)*2*PI;
        r= randn(0,1)*r0;
        pos[0][i2]=r*cos(phi1);
        pos[1][i2]=r*sin(phi1);
        pos[2][i2] = 0;
        flx[0][i2] = 0;flx[1][i2] = 0;flx[2][i2] = 1;
    }


    start = clock();//Storing clock value

    int l = 0,iii,jjj,kk,cnt;
    double **tj; //# to hold the Pinv[j](u)
```

```
50      tj = (double **)malloc(N*sizeof(double*));
51      for(int j2 = 0;j2<N;j2++){
52          tj[j2] = (double *)malloc((nj-1)*sizeof(double));
53      }
54
55
56      int *iit; //Stores the index where theta0>0.08
57      iit= (int *)malloc(N*sizeof(int));
58      int *ii; //Stores the index where status is 0.
59      ii= (int *)malloc(N*sizeof(int));
60      double sx,sy,sz,phi,theta1,cosphi,sinphi,sintheta,costheta,t2;
61      double *u = (double *)malloc(N*sizeof(double)); //To store
           random values
62      double sdep2[nj-1],ydep[nj-1];
```

In this block, all the variables listed in the previous listing are initialized to their initial values. The "pos" variable is initialized to random values. Rest of the variables are either initialized to 0 or -1 or 1. After that there are some declaration of variables which are used in the main simulation loop.

Listing 24: Initializing Seeds for rand in openacc

```
1
2  /*The below section is only meant for openacc*/
3
4      /*int nii=0,l1=0,count=0;
5      unsigned long prev = 14568725,prev1 = 4589235,m = 2147483648,
           tmp;
6      double y0,y1,y2,y3,y4,y5,y6,y7,y2_0,y2_1,y2_2,y2_3,y2_4,y2_5,
           y2_6,y2_7;
7
8      double *r_u = (double *)malloc(N*sizeof(double));
9      double *r_phi = (double *)malloc(N*sizeof(double));
10     for(i=0;i<N;i++){
11         r_phi[i]=(1.0*rand()/RAND_MAX)*2*PI;
12         r_u[i] = (1.0*rand()/RAND_MAX);
13     }
14     start = clock();//Storing clock value
15     p_st = clock();
16
17     #pragma acc data copy(r_u[:N],r_phi[:N],y2_0,y2_1,y2_2,y2_3,
```

```
        y2_4,y2_5,y2_6,y2_7,nii,prev,prev1,m,tmp,flx[0:3][0:N],↩
        intensity[0:N],status[:N],traj[:3][:ntraj][:nt],beam2[:N↩
        ][:2],theta0[:N],tj[:N][:nj-1],pos[0:3][0:N],phi0[0:N],u[:N↩
        ],ydep[:nj-1],sdep2[:nj-1]) copyin(sdepth[:nj-1],angle[0:ni↩
        ],P[0:nj][0:ni],Pinv[0:nj-1][0:n],z0,i,ntraj,x_att[:n_att],↩
        y_att[:n_att],y2_att[:n_att])
18      {
19  */
```

This section finds initial seeds in CPU and then transfer all these seeds into the GPU for parallel computation of random numbers. The above line copies all the mentioned variables in GPU and it takes around 80s.

Listing 25: Main Simulation Loop

```
1       for (k=0;k<nt;k+=100){
2           if(k+100 <= nt) kk = k+100;
3           else kk = nt;
4           //kk=min(k+100,nt);
5           for (i=k;i<kk;i++){
6               p_st = clock();
7
8               int nii=0;
9               for(int i3=0;i3<N;i3++){
10                  if(status[i3]==0){
11                      ii[nii]=i3;
12                      nii++;
13                  }
14              }
15              printf("Len of nii->%d\n",nii );
16              //# generate the random move.
17              //# Bias the rays to scatter within p radians of orig ↩
                    direction.
18
19              //The below line updates the value of i in GPU.
20              //The below line is meant only for openacc
21              //#pragma acc update device(i)
22
23              if (fscattering!=""){
24
25                  for(iii=0;iii<nii;iii++){
```

```
26                      phi0[ii[iii]]=(1.0*rand()/RAND_MAX)*2*PI;
27                      u[iii] = (1.0*rand()/RAND_MAX);
28
29                  }
```

The main simulation loop starts here. If the value of any index in status array is non-zero then it implies that ray has reached the submarine in that many steps. So we have to update the rays which are yet to reach the submarine. Hence we have to find all the index where status is zero. Therefore the first for loop traverses through status array and stores the index wherever its 0 and increases the count and stores in nii. Next in the second for loop for that many times i.e. nii times it calculates random variables and stores in u and phi0 arrays.

Listing 26: Random number in GPU

```
1  //This is how the random number are generated in GPU and is ↩
       paralleled.
2  /*
3          #pragma acc parallel loop gang private(prev,prev1)
4                      for(iii=0;iii<N;iii++){
5                          prev = m*r_u[iii];
6                          prev1 = m*r_phi[iii];
7
8                          prev = rnd(prev);
9                          u[iii] = (prev*1.0)/m;
10                         r_u[iii] = u[iii];
11
12                         prev1 = rnd(prev1);
13                         phi0[iii] = (prev1*1.0)/m;
14                         r_phi[iii] = phi0[iii];
15
16                     }
17                     */
```

The above section finds the random numbers. It does so by calling the pseudo random generator function defined earlier. prev and prev1 are the variables which are local to each threads. The function requires previous random value to generate the next random value. Hence the previous random value is stored in r_u and r_phi. Also one difference is in openacc I'm generating N random numbers instead of nii and its because I am saving

time to forego a loop which will calculate the value of nii.

Listing 27: Calculation of tj and theta0

```
/*The below directive is meant when running in openacc*/
            //#pragma acc parallel loop gang independent present(tj,P,Pinv,angle)
                for(int j2 = 0;j2<nj-1;j2++){

/*The below directive is meant when running in openacc*/
                    //#pragma acc loop vector
                    for(int temp = 0;temp<nii;temp++){
                        //Results of python code matches when j2 below is replaced by 7
                        if(u[temp]>=P[j2][1]){
                            tj[ii[temp]][j2] = splint((double *)P+j2*Nc+1,(double *)angle+1,(double *)Pinv+j2*n,Nc-1,u[temp]);
                        }
                        else tj[ii[temp]][j2] = 0;
                    }
                }
/*The below directive is meant when running in openacc*/
            //#pragma acc parallel loop gang independent private(yp1,ypn,y2_0,y2_1,y2_2,y2_3,y2_4,y2_5,y2_6,y2_7) present(tj,sdepth)
                for(int i2=0;i2<N;i2++){
                    if(status[i2] == 0){
                        yp1 = (tj[i2][1]-tj[i2][0])/(sdepth[1]-sdepth[0]);
                        ypn = (tj[i2][nj-2]-tj[i2][nj-3])/(sdepth[nj-2]-sdepth[nj-3]);

                        for(int j3 = 0;j3<nj-1;j3++){
                            ydep[j3] = tj[i2][j3];
                        }

                        spline(sdepth,ydep,nj-1,yp1,ypn,sdep2);
                        theta0[i2]=splint(sdepth,ydep,sdep2,nj-1,pos[2][i2]);


```

```
30          /* The commented part below is meant while running in ↩
               openacc. It is a substitute for the above lines till ↩
               ypn.*/
31                      /*spline1(sdepth,tj[i2][0],tj[i2][1],tj[i2↩
                           ][2],tj[i2][3],tj[i2][4],tj[i2][5],tj[i2↩
                           ][6],tj[i2][7],nj-1,yp1,ypn,&y2_0,&y2_1↩
                           ,&y2_2,&y2_3,&y2_4,&y2_5,&y2_6,&y2_7);
32                      theta0[i2]=splint1(sdepth,tj[i2][0],tj[i2↩
                           ][1],tj[i2][2],tj[i2][3],tj[i2][4],tj[i2↩
                           ][5],tj[i2][6],tj[i2][7],y2_0,y2_1,y2_2,↩
                           y2_3,y2_4,y2_5,y2_6,y2_7,nj-1,pos[2][ii[↩
                           i2]]);*/
33                  }
34
35              }
36          }
```

In the first for loop, depending on the value of random variable "u", it calls the splint function or else assigns 0 to tj. Now the discrepancy faced here is that it gives different results in python and c. If I plot one non-zero row of tj in python, it is always constant but in C, the value differs, its kind of hyperbola. But as the curve ends the values become nearly equal in C and python. Therefore while making spline calls, if I always call the last spline curve (index = 7) irrespective of the index(j2) the results of the simulation matches with those in Python. If the index is fixed, the values of tj rows are constant and hence there is no problem in extrapolation but if it isn't then extrapolation gives wrong results and hence when extrapolating, we assume a constant value. The plots are attached below. Now, in the second for loop, as tj is 2-D array, for each row we will call spline function and store the 2nd order derivative in sdep2. Then we will call the splint function to find the corresponding value to pos[2] array and store the output in theta0.

Listing 28: Updating of theta0vals

```
1           if(i%10==0){
2               int l1;
3
4               int count=0;
5               for(int iii=0;iii<N;iii++){
6                   if(theta0[iii]>0.08){
7                       iit[count]=iii;
8                       count++;
```

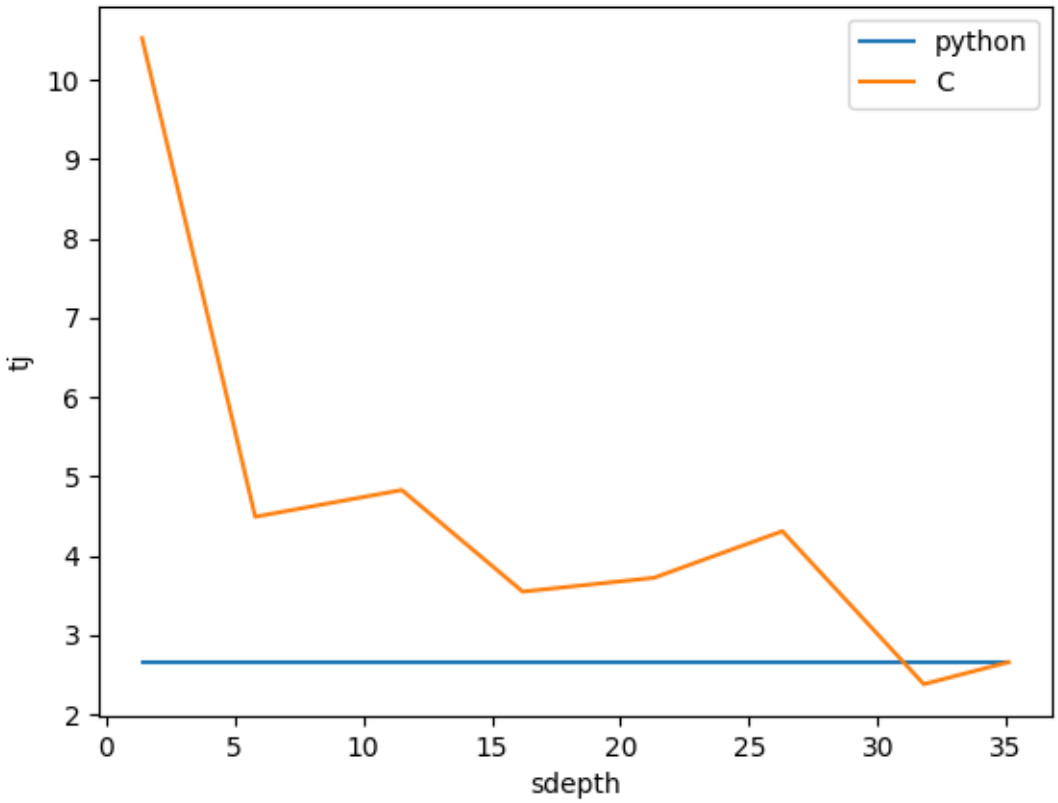


Figure 7: Plot of a row of tj vs sdepth

```
9                   }
10              }
11              printf("len(iit)=%d\n",count);
12              if(count>1000) l1=1000;
13              else l1=count;
14
15              for(count = l;count<l+l1;count++){
16                  theta0vals[count][1]=theta0[iit[count-l]];
17                  theta0vals[count][0]=pos[2][iit[count-l]];
18              }
19
20              l+=l1;
21          }
```

This block runs once in every 10 iterations. First we store the index wherever theta0>0.08 in iit variable and increment count whenever the condition is true. Based on the count value whether it is more than 1000 or not, we update theta0vals. At every 10th iteration, we store 1000 values of theta and pos[2] array. The variable l represents the index till

where theta0vals has been filled. But this part is not used anywhere and hence its been removed while porting it into openacc.

Listing 29: Updating pos flx and intensity

```
/*The below directive is meant when running in openacc*/
            //#pragma acc parallel loop gang independent private(sx,sy,sz,phi,theta1,cosphi,costheta,sinphi,sintheta)
                for(iii=0;iii<N;iii++){
                    if (status[iii] == 0){


                        sx= cos(phi0[iii])*sin(theta0[iii]);
                        sy= sin(phi0[iii])*sin(theta0[iii]);
                        sz= cos(theta0[iii]);
                        phi=atan2(flx[1][iii],flx[0][iii]);
                        theta1=atan2(sqrt(pow(flx[0][iii],2)+pow(flx[1][iii],2)),flx[2][iii]);
                        cosphi=cos(phi);sinphi=sin(phi);
                        costheta=cos(theta1);sintheta=sin(theta1);

                        flx[0][iii]= (cosphi*costheta*sx) + (-sinphi*sy) + cosphi*sintheta*sz;
                        flx[1][iii]= (sinphi*costheta*sx) + (cosphi*sy) + sinphi*sintheta*sz;
                        flx[2][iii]= (-sintheta*sx) + costheta*sz;
                        intensity[iii]=intensity[iii]*exp(-splint(x_att,y_att,y2_att,n_att,pos[2][iii]));

                        pos[0][iii] = pos[0][iii]+flx[0][iii];
                        pos[1][iii] = pos[1][iii]+flx[1][iii];
                        pos[2][iii] = pos[2][iii]+flx[2][iii];
                        if(pos[2][iii] < 0) pos[2][iii] = pos[2][iii]*(-1);
                    }

                }
```

This is the loop where the position and intensity of the rays are updated based on some calculations. Intensity decreases in an exponential way. Based on the attenuation csv file, we had drawn a spline fit and using splint function, we update the intensity of the rays.

In python code, sx, sy, sz, phi, etc are all arrays but are not used anywhere except for updating. Hence to save memory, its defined as a double variable.

Listing 30: Updating Status and traj

```
/*The below directive is meant when running in openacc*/
            //#pragma acc parallel loop gang private(t2)
                for(int i2=0;i2<N;i2++){
                    if(status[i2] == 0){
                        t2 = sqrt(pos[0][i2]*pos[0][i2]+pos[1][i2]*↩
                           pos[1][i2]+(pos[2][i2]-z0)*(pos[2][i2]-↩
                           z0));
                        if(t2<=s){
                            status[i2] = i;
                        }
                    }
                }
            //# save trajectories

/*The below directive is meant when running in openacc*/
            //#pragma acc parallel loop gang independent collapse↩
               (2) present(pos,traj)
                for(int tt = 0;tt<3;tt++){
                    for(int tp = 0;tp<ntraj;tp++){
                        traj[tt][tp][i]=pos[tt][tp];
                    }
                }
```

In the first loop, we find the norm and store it in the variable t2. Now if t2 <= s, we update status array i.e. that ray has reached the submarine in i steps. Therefore the value of status of that ray is set to i, which allows to find the average steps taken to reach submarine.
In the second for loop we store the entire pos array into traj array which allows in plotting ray trajectories.

Listing 31: Updating beam1 and beam2

```
            /*# not a good algorithm: assumes rays with (0,0) have ↩
               not yet reached corresponding depth.
            # save rays crossing z1*/
/*The below directive is meant when running in openacc*/
```

```
4               //#pragma acc parallel loop gang independent present(↩
                   pos,beam1)
5                for(int tt = 0;tt<N;tt++){
6                    if((pos[2][tt] > z1) && (beam1[tt][0]<0)){
7                        beam1[tt][0]=sqrt(pos[0][tt]*pos[0][tt]+pos↩
                            [1][tt]*pos[1][tt]);
8                        beam1[tt][1]=atan2(pos[0][tt],pos[1][tt]);
9                    }
10               }
11
12
13              /*# save rays crossing z2
14              */
15
16  /*The below directive is meant when running in openacc*/
17              //#pragma acc parallel loop gang independent present(↩
                   pos,beam2)
18               for(int tt = 0;tt<N;tt++){
19                   if((pos[2][tt] > z2) && (beam2[tt][0]<0)){
20                       beam2[tt][0]=sqrt(pos[0][tt]*pos[0][tt]+pos↩
                            [1][tt]*pos[1][tt]);
21                       beam2[tt][1]=atan2(pos[0][tt],pos[1][tt]);
22                   }
23               }
```

In python xy2polar function was defined and both beam1 and beam2 called the same function. In C, we have eliminated the function and updated the value of beam1 and beam2 then and there based on the value of pos[2] array whether is more than z1 or z2 respectively. But again I couldn't find any use of beam1. It is only updated inside the loop and not used for any plots or calculations.

Listing 32: End of main Simulation Loop

```
1               end = clock();
2               cpu_time_used = ((double) (end - start)) / ↩
                   CLOCKS_PER_SEC;
3               double cpu_time_used1 = ((double) (end - p_st)) / ↩
                   CLOCKS_PER_SEC;
4               printf("Loop - %d (%lf sec) loop time (%lf sec)\n\n\n",↩
                   i,cpu_time_used,cpu_time_used1);
5
```

```
        }
        //# write out iter number every 100 iterations
        //time2=time.time()
        end = clock();
        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("Loop - %d (%lf sec)\n",k,cpu_time_used);


        //@@What is this doing.
        //sys.stdout.flush();// # force immediate write

    }
    free(iit);free(ii);
```

This marks the end of the main simulation loop. After every iteration it prints the time taken to complete that loop. Also it deallocates the memory of ii and iit.

Listing 33: Calculation of Pvals

```
    double pvals[nj-1][Nc-1];
    for(i=0;i<nj-1;i++){
        for(j=0;j<ni-1;j++){
            pvals[i][j] = splint(sangle,(double *)Sca_T+ni*(i+1)↩
               +1,(double *)sc+(i)*(ni-1),n,sangle[j]) * 180.0/PI;
        }
    }
```

This loop fills the pvals matrix, which is used in plotting scattering profile vs depth. In python there is a function named p but in C, we have replicated it using two loops and splint calls.

Listing 34: Storing Results

```
    printf("Storing Variables -> status,theta0vals,flx,intensity,↩
       tsrc,channel,traj,beam2,pvals");
    printf("Constant Variables -> rmax2,ntraj,nbins2,nt,sdepth,↩
       sangle");

    FILE *fptr;
    fptr = fopen("st.csv", "w");
    if(fptr == NULL){
```

```
7           printf("Error!");
8           exit(1);
9       }
10      fprintf(fptr,"#Status\n");
11      for(i=0;i<N;i++){
12          fprintf(fptr, "%d",status[i]);
13          if(i!=(N-1)) fprintf(fptr, ",");
14      }
15
16      fclose(fptr);
17      fptr = fopen("pvals.csv", "w");
18      for(i=0;i<nj-1;i++){
19          for(j=0;j<Nc-1;j++){
20              fprintf(fptr, "%lf",pvals[i][j]);
21              if(j!=Nc-2) fprintf(fptr, ",");
22          }
23          fprintf(fptr, "\n");
24      }
25
26
27      fclose(fptr);
28      fptr = fopen("th.csv", "w");
29      fprintf(fptr, "#theta0vals - Transpose\n");
30
31      for(i=0;i<2;i++){
32          for(j=0;j<a2;j++){
33              fprintf(fptr, "%lf",theta0vals[j][i]);
34              if(j!=a2-1) fprintf(fptr, ",");
35          }
36          fprintf(fptr, "\n");
37      }
38
39      fclose(fptr);
40      fptr = fopen("flx.csv", "w");
41      fprintf(fptr, "#flx\n");
42      for(i=0;i<3;i++){
43          for(j=0;j<N;j++){
44              fprintf(fptr, "%lf",flx[i][j]);
45              if(j!=N-1) fprintf(fptr, ",");
46          }
47          fprintf(fptr, "\n");
```

```
48      }
49
50      fclose(fptr);
51      fptr = fopen("in.csv", "w");
52      fprintf(fptr, "#intensity\n");
53      for(i=0;i<N;i++){
54          fprintf(fptr, "%lf",intensity[i]);
55          if(i!=N-1) fprintf(fptr, ",");
56      }
57
58      fclose(fptr);
59      fptr = fopen("tsrc.csv", "w");
60      fprintf(fptr, "#tsrc\n");
61      for(i=0;i<N;i++){
62          fprintf(fptr, "%d",tsrc[i]);
63          if(i!=N-1) fprintf(fptr, ",");
64      }
65
66      fclose(fptr);
67
68      fptr = fopen("beam.csv", "w");
69      fprintf(fptr, "#beam2 - Transpose\n");
70
71      for(i=0;i<2;i++){
72          for(j=0;j<N;j++){
73              fprintf(fptr, "%lf",beam2[j][i]);
74              if(j!=N-1) fprintf(fptr, ",");
75          }
76          fprintf(fptr, "\n");
77      }
78
79      fclose(fptr);
80      fptr = fopen("traj.csv", "w");
81      fprintf(fptr, "#traj\n");
82
83      for(i=0;i<3;i++){
84          for(j=0;j<10;j++){
85              for(k=0;k<nt;k++){
86                  fprintf(fptr, "%lf",traj[i][j][k]);
87                  if(k!=nt-1) fprintf(fptr, ",");
88              }
```

```
89              fprintf(fptr, "\n");
90          }
91          fprintf(fptr, "\n\n");
92      }
93      fclose(fptr);
94
95      for(int i=0;i<3;i++){
96          free(flx[i]);
97          free(direction[i]);
98          free(pos[i]);
99      }
100     //printf("Coming here 3\n");
101     free(direction);
102     free(flx);
103     free(pos);
104     free(status);
105     free(intensity);
106     free(tsrc);
107     //printf("Coming here 1\n");
108
109     for(int i=0;i<3;i++){
110         //traj[i] = (double**)malloc(sizeof(double*)*ntraj);
111         for(int j=0;j<ntraj;j++){
112             free(traj[i][j]);
113         }
114     }
115     for(int i=0;i<3;i++){
116         free(traj[i]);
117     }
118     free(traj);
119
120
121     for(int i=0;i<N;i++){
122         free(beam2[i]);
123         free(beam1[i]);
124         free(tj[i]);
125     }
126     free(beam2);
127     free(beam1);
128     free(tj);
129
```

```
130
131     //printf("Coming here2\n");
132     for(int i=0;i<a2;i++){
133         free(theta0vals[i]);
134     }
135     free(theta0vals);
136
137     free(phi0);
138     free(theta0);
139     free(u);
140
141
142     printf("Done\n");
143     return 0;
144 }
```

This section marks the end of the code. All the necessary variables which are needed for plotting are stored in excel files. A python code then reads all the files and do the necessary plotting. The python code is listed below. Also all the memory of the dynamically allocated variables are freed.

B Python Code for plotting

AS said above, python is good for plotting and therefore all the plotting are done using python. So to do the same, all the variables are stored in the csv file and plotted using python code.

Listing 35: Code for plotting: Pre-Processing

```
1  from pylab import *
2  import mpl_toolkits.mplot3d.axes3d as p3
3  import sys
4  #from scipy import weave # not used now
5  from scipy.integrate import quad
6  import time
7  from scipy.interpolate import UnivariateSpline
8
9  N = 100000
10 nt=250  # number of time steps to simulate. By this time, active
11 z0=100  # centre of submarine
12 s=10    # size of submarine (cube)
13 ntraj=10 # number of trajectories to track for plotting
14 seed() # randomize the random number generator
15 r0=5   # size of the input beam region
16 wavelength=513 # nanometers
17 # file containing attenuation information
18 fattenuation="data/absorption_coefficient_april2017_T3D3.csv"
19 # fattenuation=""
20 attnConst=0.001
21 # if file is an empty string, attenuation is a fixed number, given ↩
      by attnConst
22
23 fscatname="VSF-sept2017-T2D3"
24 # fscatname="VSF-sept2017-T2D1"
25 fscattering="data/"+fscatname+".csv"
26
27 Nc=99
28 fsflag="scatdata"
29 Scb=loadtxt(fscattering,delimiter=",")
30 ni,nj=Scb.shape
31
32 nj=9
```

```
33 ##################################################
34 Sca=Scb[:Nc,:nj] # truncate to nj columns
35 ni=Nc
36 del Scb
37 # Sca has scattering data. Column 0 contains the angles and
38 # row zero contains the depth values. The rest are the
39 # scattering data for that depth, angle combination.
40 sdepth=Sca[0,1:]
41 sangle=Sca[1:,0]       # in degrees
42 stheta=sangle*pi/180.0 # in radians
43 dtheta=zeros(stheta.shape)
44 dtheta[1:-1]=(stheta[2:]-stheta[:-2])*0.5
45 r2deg=180.0/pi
46 angle=zeros(ni)
47 angle[1:]=sangle # data does not have theta=0. angle has this.
48 theta=angle*pi/180.0
49
50 z1=0
51 z2=z0-2*s
52 rmax2=10000
53 nbins2=100
54
55 P = loadtxt("P.csv",delimiter=",")
56 status = loadtxt("st.csv",delimiter=",")
57 flx = loadtxt("flx.csv",delimiter=",")
58 theta0vals = loadtxt("th.csv",delimiter=",")
59 beam2 = loadtxt("beam.csv",delimiter=",")
60 temp = loadtxt("traj.csv",delimiter=",")
61 #channel = loadtxt("ch.csv",delimiter=",")
62 pvals = loadtxt("pvals.csv",delimiter=",")
63 channel=zeros(nt)
64 tsrc = loadtxt("tsrc.csv",delimiter=",")
65 intensity = loadtxt("in.csv",delimiter=",")
66 beam2 = beam2.T
67 theta0vals = theta0vals.T
68
69 traj=zeros((3,ntraj,nt))
70 traj[0,:,:] = temp[:10,:]
71 traj[1,:,:] = temp[10:20,:]
72 traj[2,:,:] = temp[20:,:]
```

Here it reads the excel file, declare and initialize some variables. Then it reads all the output files created by the C code and do some necessary computations before plotting.

Listing 36: Code for plotting: Post-Processing

```
1
2  fname = "test"
3
4  # post processing
5  print ("\n\n%d rays (out of %d) reached the submarine" % (len(where↩
       (status>0)[0]),N))
6  z0max=int(theta0vals[0,:].max())
7  for zz in range(z0max):
8          iii=where( abs(theta0vals[0,:]-zz)<=0.5 )[0]
9          print (len(iii))
10         theta0vals[1,iii[1000:]]=-1
11
12
13 z=zeros((ni-1,nj-1))
14
15 figure(7)
16 for j in range(1,nj):
17     loglog(sangle,Sca[1:,j])
18     z[:,j-1]=Sca[1:,j]
19
20 title("Scattering profile data")
21 xlabel(r"$\theta$",size=16)
22 name = fname+"scat-profile.png"
23 savefig(name)
24
25
26 figure(8)
27 w=log10(abs(z.T))
28 w[w<-5]=-5
29 contourf(sangle,sdepth,w[-1::-1],[-5,-2,-1,0,0.5,1,1.5,2])
30 colorbar()
31 xlabel(r'$\theta$',size=16)
32 ylabel(r'$d$',size=16)
33 title(r"measured scattering data")
34 name = fname+"-scat-contour.png"
35 savefig(name)
36
```

```
37
38
39 figure(10)
40 for j in range(nj-1):
41     plot(angle,P[j,:])
42
43 title("CDF of scattering data")
44 xlabel(r"$\theta$ (degrees)",size=16)
45 print ("Done quad block")
46 name = fname+"-CDF.png"
47 savefig(name)
48
49
50 figure(6)
51 plot(sdepth,pvals,'b',lw=3)
52 xlabel(r"$z$",size=20)
53 ylabel(r"$\sigma_p$",size=20)
54 xlim([0,z0]);xticks(size=20)
55 ylim([0,45]),yticks(size=20)
56 title("Scattering profile vs. depth")
57 grid(True)
58 name = fname+"-pvsz.png"
59 savefig(name)
60
61
62 figure(2)
63 ll=where(status>0)[0] # find rays that hit submarine
64 sphi=arctan2(flx[1,ll],flx[0,ll])
65 r2=flx[0,ll]**2+flx[1,ll]**2
66 subtheta=arccos(flx[2,ll]/sqrt(r2+flx[2,ll]**2))
67 nbins1=sqrt(len(ll))
68 factor=180/pi
69 hist(subtheta*factor,arccos(linspace(1,-1,nbins1))*factor)
70 title(r"histogram of ray directions in $\theta$")
71 xlabel(r"$\theta$ (degrees)")
72 name = fname+"-raydir.png"
73 savefig(name)
74
75 # figure(6)
76 nbins=sqrt(len(ll))/10
77 x=linspace(-pi,pi,nbins)
```

```
78 y=arccos(linspace(1,-1,nbins))
79 # hist2d(sphi,subtheta,[x,y])
80 # title(r"histogram of ray directions. Note $\theta=0$ is bottom ↩
      surface")
81 # xlabel(r"$\phi$")
82 # ylabel(r"$\theta$ (non-uniform bins)")
83 ll=where(status>0)[0]
84 tof=(status[ll]-tsrc[ll])
85 #figure(3)
86 #hist(tof*5e-3,100,log=True)
87 #title("Histogram of ray time of flight")
88 #xlabel(r"time ($\mu$sec)")
89 #savefig(fname+"-tof.png")
90 print ("Average time to reach submarine=%.2f time steps" % (mean(↩
      tof)))
91 print ("Stdeviation of time to reach submarine=%.2f time steps" % (↩
      std(tof)))
92 for i in range(nt):
93     mm=where(tof==i)[0]
94     channel[i]=sum(intensity[mm])
95
96 figure(4)
97 plot(arange(nt)*5e-3,channel,lw=3)
98 xlim([0,12])
99 title("Intensity Channel as predicted by simulation")
100 xlabel(r"time ($\mu$sec)",size=20)
101 grid(True)
102 name = fname+"-channel.png"
103 savefig(name)
104
105 # We finally plot a few actual ray trajectories.
106 fig5=figure(5)
107 bx=p3.Axes3D(fig5)
108 for i in range(ntraj):
109     bx.plot3D(traj[0,i,:],traj[1,i,:],traj[2,i,:])
110 title("ray trajectories")
111 grid(True)
112 name = fname+"-traj.png"
113 savefig(name)
114
115 # Histogram at z2. Bins are equal area bins.
```

```
116 fig7=figure(14)
117 ii=find(beam2[:,0]>=0) # find all rays that crossed z2
118 locs=sqrt(linspace(0,rmax2,nbins2))
119 hist(beam2[ii,0],locs)
120 grid(True)
121 title("Histogram of rays at $z=%d$ metres" % z2)
122 xlabel(r"$r$",size=16)
123 name = fname+"-beam.png"
124 savefig(name)
125 show()
```

This is the main block which plots all the files.